# NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

# THESIS

## CLOUD FINGERPRINTING: USING CLOCK SKEWS TO DETERMINE CO-LOCATION OF VIRTUAL MACHINES

by

Christopher J. Wasek

September 2016

Thesis Co-Advisors: Geoffrey G. Xie
Mathias Kolsch

**Approved for public release. Distribution is unlimited.**

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

*Form Approved OMB No. 0704–0188*

*Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.*

| | | |
|---|---|---|
| 1. AGENCY USE ONLY *(Leave Blank)* | 2. REPORT DATE<br>September 2016 | 3. REPORT TYPE AND DATES COVERED<br>Master's Thesis 3/19/2015 - 9/17/2016 |

4. TITLE AND SUBTITLE

CLOUD FINGERPRINTING: USING CLOCK SKEWS TO DETERMINE CO-LOCATION OF VIRTUAL MACHINES

5. FUNDING NUMBERS

6. AUTHOR(S)

Christopher J. Wasek

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Cloud computing has quickly revolutionized computing practices of organizations, to include the Department of Defense. However, security concerns over co-location attacks have arisen from the consolidation inherent in virtualization and from physical hardware hosting virtual machines for multiple businesses and organizations. Current cloud security methods, such as Amazon's Virtual Private Cloud, have evolved defenses against most of the well-known fingerprinting and mapping methods in order to prevent malicious users from determining virtual machine co-location on the same hardware. Our solution to co-locating virtual machines unhindered was to derive their clock skews, or the temporal deviation of the system clock over time. Capturing normal TCP traffic to analyze timestamps from a virtual machine in the cloud, our results were inconclusive in demonstrating that co-located virtual machines will have similar clock skews due to large, inconsistent packet delays. Our research demonstrates a potential vulnerability in cloud defenses so that cloud users and providers can take appropriate steps to prevent malicious co-location attacks.

14. SUBJECT TERMS

cloud, TCP timestamps, clock skews, side-channel attacks, virtual machines, VM co-location, fingerprinting

15. NUMBER OF PAGES 85

16. PRICE CODE

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
| Unclassified | Unclassified | Unclassified | UU |

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release. Distribution is unlimited.**

# CLOUD FINGERPRINTING: USING CLOCK SKEWS TO DETERMINE CO-LOCATION OF VIRTUAL MACHINES

Christopher J. Wasek
Lieutenant Commander, United States Navy
B.S., U.S. Naval Academy, 2006

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**September 2016**

Approved by: Geoffrey G. Xie
Thesis Co-Advisor

Mathias Kolsch
Thesis Co-Advisor

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

Cloud computing has quickly revolutionized computing practices of organizations, to include the Department of Defense. However, security concerns over co-location attacks have arisen from the consolidation inherent in virtualization and from physical hardware hosting virtual machines for multiple businesses and organizations. Current cloud security methods, such as Amazon's Virtual Private Cloud, have evolved defenses against most of the well-known fingerprinting and mapping methods in order to prevent malicious users from determining virtual machine co-location on the same hardware. Our solution to co-locating virtual machines unhindered was to derive their clock skews, or the temporal deviation of the system clock over time. Capturing normal TCP traffic to analyze timestamps from a virtual machine in the cloud, our results were inconclusive in demonstrating that co-located virtual machines will have similar clock skews due to large, inconsistent packet delays. Our research demonstrates a potential vulnerability in cloud defenses so that cloud users and providers can take appropriate steps to prevent malicious co-location attacks.

THIS PAGE INTENTIONALLY LEFT BLANK

# Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Figures

# List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Acronyms and Abbreviations

| | |
|---|---|
| **ACL** | Access Control List |
| **AWS** | Amazon Web Services |
| **DOD** | Department of Defense |
| **DNS** | Domain Name Service |
| **EC2** | Elastic Cloud Computing |
| **EoR** | End of Row |
| **GCE** | Google Compute Engine |
| **IaaS** | Infrastructure-as-a-Service |
| **ICMP** | Internet Control Message Protocol |
| **IP** | Internet Protocol |
| **IT** | Information Technology |
| **NPS** | Naval Postgraduate School |
| **NTP** | Network Time Protocol |
| **OS** | operating system |
| **OLS** | Ordinary Least Squares |
| **PaaS** | Platform-as-a-Service |
| **RANSAC** | Random Sample Consensus |
| **RTT** | round trip time |
| **SaaS** | Software-as-a-Service |
| **TCP** | Transmission Control Protocol |

|  |  |
|---|---|
| **ToR** | Top of Rack |
| **TSecr** | Timestamp Echo Reply |
| **TSopt** | Timestamps option |
| **TSval** | Timestamp Value |
| **VM** | virtual machine |
| **VPC** | Virtual Private Cloud |
| **VPN** | Virtual Private Network |
| **Win7** | Windows 7 |
| **Win10** | Windows 10 |

# Acknowledgments

The last year has been a roller coaster ride as I began the journey that has culminated in this thesis. Throughout this journey, I have received immense support from my family, friends, and NPS professors. Professor Xie, your knowledge and enthusiam for networking is what steered me towards this topic in the beginning and helped get me through the detailed understanding and simulation of TCP timestamps and network behavior. Professor Kolsch, without your guidance my understanding of regression modeling, the AWS cloud, and (life-saving) automated deployment of instances would not have been anywhere near the level necessary for this study. You both have been instrumental in my work progression and for that I cannot thank you enough.

My father, Dr. James Wasek, you provided advice and many last minute critiques as I worked to make deadlines. You have always been an inspiration to me and I would not be who I am today without you.

And finally my wife, Sascha, and my children, Zoe and Liam, it is for you that I have accomplished this feat. You all have sacrificed greatly over the last 18 months while giving me your full support, providing an ear to vent to, and being my rock to lean on. I thank you and love you all from the bottom of my heart!

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 1:
# Introduction

Use of the cloud has quickly become the way of the future in computing by organizations, to include the Department of Defense (DOD). Whether it is through Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), or Software-as-a-Service (SaaS), utilization of a third-party business such as Microsoft Azure, Google Compute Engine (GCE), and Amazon Elastic Cloud Computing (EC2), allows organizations to eliminate the purchase, maintenance, and administration of their own server infrastructures and save on Information Technology (IT) expenses. However, because the cloud infrastructure is located off-site from an organization and the cloud is available to anyone who pays for its services, cloud security has become a primary concern for its many users.

## 1.1 Proliferation of Cloud Computing

As computers continue to merge into every aspect of our life, the sheer quantity of data and applications has grown at an exponential rate. In response, cloud computing has quickly become the de facto method for both managing and processing this data [1]. The high demand for cloud services has caused many companies to offer easy solutions at cheap prices while still gaining significant profits. In fact, current forecasts show continual growth in this market over the next ten years [2]. For example, in the last quarter of 2015 alone, Columbus [2] reports Amazon Web Services (AWS) "generated $7.88B in revenue, up 69% over last year" while overall consumer spending for IaaS services in 2016 is anticipated to reach $38B and $173B in 2026. The projected annual spending costs for public cloud IaaS, PaaS, and SaaS from now until 2026 are illustrated in Figure 1.1. Cloud computing offers many advantages and is easy to use; the demand for it will only increase.

## 1.2 Problem Statement

Co-location attacks pose great risks to all legitimate users of the cloud, especially organizations like the DOD that could potentially store sensitive files and data on cloud servers. An emergent threat with the increased use of cloud computing, a **co-location attack** is

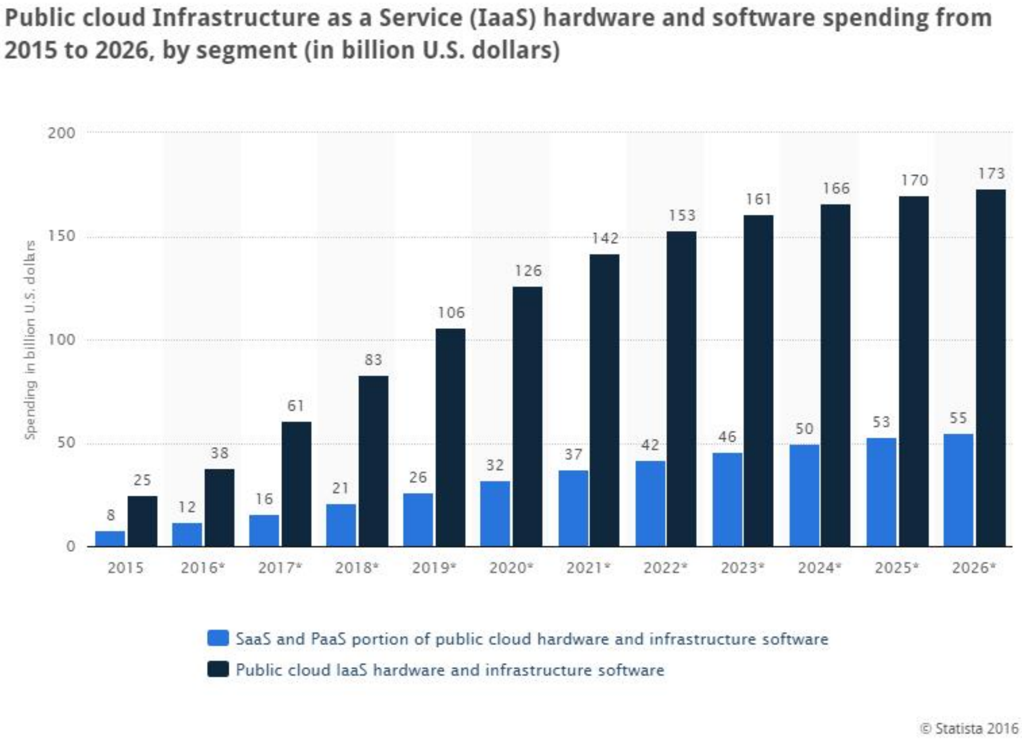


Figure 1.1: Ten Year Forecast of Consumer Spending on Public Cloud Services. Source: [2].

conducted by a malicious user setting a cloud-based virtual machine (VM) to attack other VMs residing on the same physical server. Typically, this attack is designed to either deny service to that server or extract privileged information, such as Personally Identifiable Information and cryptographic keys [3]. Since cloud providers open their services to all users who are willing to pay, it is impossible to know who is sharing a server with whom. For cloud users, co-location attacks are unpredictable and an attack is detectable usually only after it has been conducted. Some cloud providers will allow the reservation and dedication of specific physical servers, AWS calls them Dedicated Instances for example [4], which reserves a physical server for a single user to launch all VMs onto. This helps with both VM computing performance (i.e., load-balancing and inter-VM communications) and security, eliminating the threat of a co-location attack. However, the large increase in fees associated with dedicated instances is often too costly for many organizations and therefore viewed as unnecessary.

In an effort to combat the threat of co-location attacks, both cloud providers and users have deployed various security methods to prevent malicious users from determining VM co-location through known fingerprinting and network mapping techniques. For example, to prevent tracerouting most users do not allow their VMs to reply to Internet Control Message Protocol (ICMP) echo requests. However, there are fingerprinting methods that are unobtrusive and appear as legitimate network traffic that can slip past today's cloud security models [3], [5], [6], [7].

Prior research conducted in [8] has shown that physical computing devices can be remotely fingerprinted by their clock skews, or the rate at which the device clock drifts compared to real time, derived from timestamps in Transmission Control Protocol (TCP) and ICMP packets. Our study focuses on the problem of determining if VMs share the clock skew fingerprint from their host server. Our assumption is that since the VM relies on the physical and virtual clocks of the host server, all VMs on a single server will exhibit similar clock skews and thereby provide evidence of co-location. We plan to utilize TCP timestamps only under the consideration that this protocol is less likely to be blocked or restricted than ICMP in a real-world cloud environment.

## 1.3 Research Questions

A number of studies have been conducted into the problem of determining co-location of an adversary VM with a target VM in cloud environments. While previous studies have allowed cloud providers to adapt methods and policies to help prevent successful use of these exposed methods, detection techniques continue to evolve in order to circumvent or exploit current cloud security constraints and practices. Our contribution to cloud VM co-location research is in analyzing the clock skews of cloud VMs. Specifically, we ask:

- Can the estimation of clock skews obtained from TCP timestamps help to accurately determine co-location of VMs in the cloud?

To the best of our knowledge, this technique has not yet been researched. Through our study of VM clock skews, we look to answer the following additional research questions:

- How many timestamps should be collected in order to reliably estimate the clock skew?

- What estimator method is most reliable at determining clock skews?
- Does the volume of network traffic influence our ability to measure a VM's clock skew?

## 1.4 Thesis Organization

The organization of this thesis is as follows: Chapter 1 introduced the problem of emergent co-location attacks within a cloud environment. We posted our central research questions and outline the thesis paper. Chapter 2 begins by discussing the fundamentals of cloud architecture and security with particular emphasis on AWS. We then describe the intent and methodology of co-location attacks. Lastly, we discuss previous work researching methodologies of determining VM co-location in the cloud to include clock skew fingerprinting. Chapter 3 investigates the implementation of clock skew modeling within a controlled lab environment. We extend our analysis into a controlled multi-server environment to validate our primary research assumption. We also compare our estimators in a simulation with a known skew under different network traffic scenarios. Chapter 4 details our methodology of testing for VM co-location in the AWS GovCloud environment and explains the analysis of our testing results. Lastly, in Chapter 5 we present our conclusions and highlight potential avenues for future work.

# CHAPTER 2:
# Background

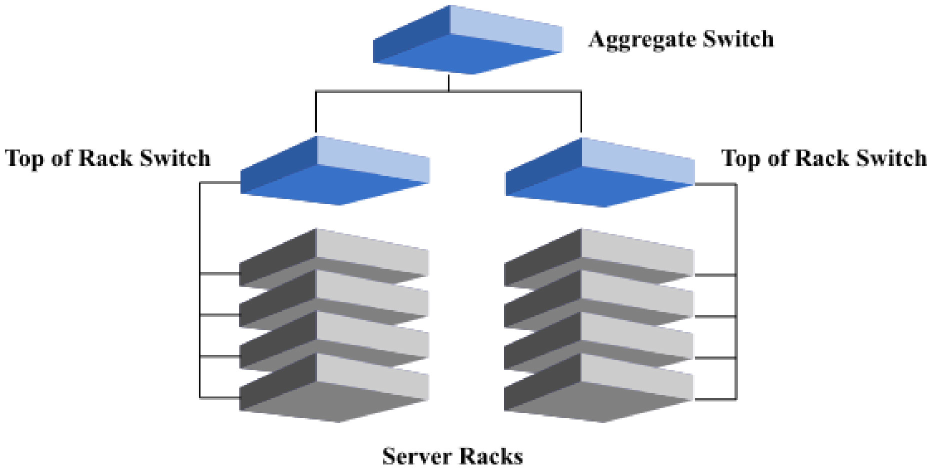
In this chapter, we begin by providing a high-level overview of a typical cloud architecture. We then discuss security practices in the cloud while providing some additional insight on the AWS Virtual Private Cloud (VPC) concept. The idea of co-location attacks will then be introduced as well as previously researched methods of co-location detection. Lastly, we detail the methodology and previous work of clock skew fingerprinting.

## 2.1 Cloud Architecture

To most, “the cloud” is complex, abstract, and intangible. It exists, but more as an idea than a thing, even as it allows users to have the flexibility to do whatever is necessary to complete their task, such as building a simple data storage system or an entire virtualized network. However, while complex and abstract in some ways, the cloud is still a physical construct. In this section, we describe the basic cloud architecture design by first looking at simple models for both the physical and logical topologies necessary to make it work. Next, we discuss the role of hypervisors, the software that manages VMs on a physical server, in a virtual environment. Last, we briefly discuss the capabilities and benefits of virtualized computing.

### 2.1.1 Topology

Topology for the cloud can be referenced two ways: the physical topology of the servers and networking gear and the logical topology as viewed from a cloud user. Physically, the cloud is simply a data center, a large cluster of servers networked together, whose singular purpose is to provide scalable VMs on demand [9]. In this regard, the topology is fairly simple. Starting from the bottom up, a VM is hosted by a hypervisor which resides on a physical server. This server is one of many servers within a rack, which is connected to an edge switch. The edge switch routes traffic from the server through any number of aggregate switches (used for hierarchical routing/switching relationships) before reaching the gateway router. It is typically classified as either a Top of Rack (ToR) switch or End of Row (EoR) switch, with the difference between the two defined as how the server racks are assigned to the switch [7]. If each switch has its own individual rack, then the switch is considered to be a ToR switch. A switch that connects servers from multiple racks is considered to be an EoR switch. A general concept of the ToR topology in illustrated in Figure 2.1. In practice, physical network connections are typically redundant in order to maintain service availability and load balancing performance [9]. For example, while an edge switch connects to one aggregate switch for primary routing, it may also be connected to a second aggregate switch in order to shift routing paths if the primary aggregate switch fails or becomes too congested with other traffic.



This depicts a simple ToR topology model found in data center architectures. Each stack of four servers represents a single server rack with all servers in each rack connected to its own ToR, or edge, switch. The two ToR switches then connect to an aggregate switch, which in turn connects to a gateway border router.

Figure 2.1: Simple Cloud ToR Topology Model

From a logical standpoint, the topology of the cloud is entirely dependent on the cloud construct that a user desires and implements, namely whether it is an IaaS, PaaS, or SaaS model. An IaaS model provides the user with vendor support only for the hardware/infrastructure necessary to run his cloud environment and store data such as the physical servers, switches, and routers. The user in this case has the freedom to build his cloud environment as he sees fit. This includes building a virtual network hiding behind a Virtual Private Network (VPN) gateway (an entrypoint to a network through an encrypted routing tunnel from an authenticated host) with configured virtual switching and routing [10]. For the PaaS model, the cloud vendor provides not just the infrastructure but also the underlying software required to run intended programs and applications. In this case, users do not see a network topology at all but rather a virtual server or database with a pre-installed operating system (OS) and other software programs and have the ability to inject executable source code, such as Java or Ruby [11]. Lastly, if a user selects the SaaS model, the cloud vendor supplies and supports everything for the user with the exception of some application configuration and non-privileged user administration. This is similar to hosting a website, where the cloud topology as viewed by the user is simply a single program or application [12]. Figure 2.2 illustrates a simple breakdown of the three cloud service models.

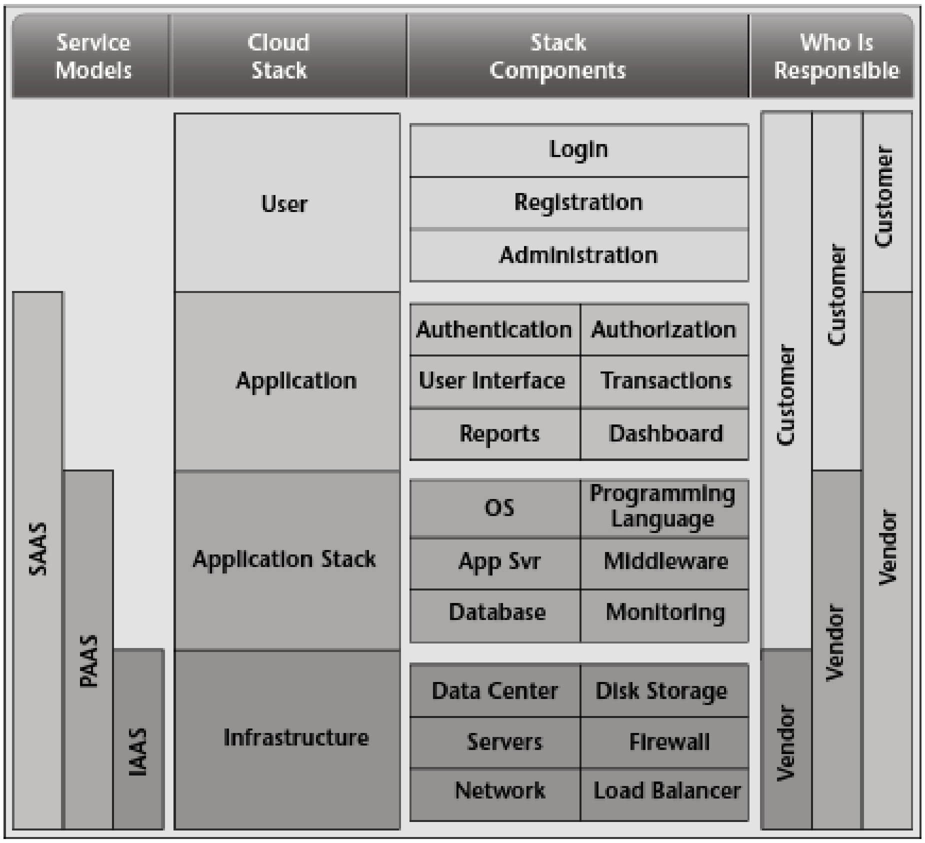


Figure 2.2: Logical Cloud Stack Architecture. Source: [12].

### 2.1.2 Hypervisors

To run a VM requires the use of a controlling system that works as a middle-man between the VM and the physical machine, known as a hypervisor. Hypervisors are sorted into two separate groups, Type I and Type II. A Type I hypervisor is a software environment that does not require support from an OS and can thus run on the bare metal hardware, such as ESXi and Xen [13]. These hypervisors are typically seen in data centers and server farms where users normally only need to interact with the hosted VMs or the hypervisor itself to manage the hosted VMs. On the other hand, a Type II hypervisor, such VirtualBox or VMWare, requires the support of a native OS [14]. These hypervisors are typically seen on laptops and workstations where a native OS already resides and the VM is used for specific purposes instead of the primary source of computing. Figure 2.3 shows a simple illustration highlighting the difference in the architecture between Type I and Type II environments.

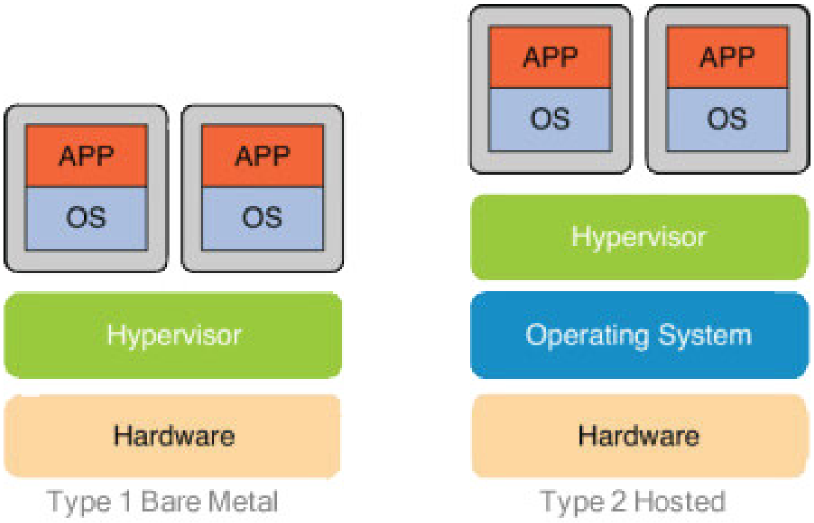


Figure 2.3: Basic Architecture of Type I and Type II Hypervisors. Source: [15].

### 2.1.3 Virtualized Computing

Virtualized computing is the mimicking of hardware computation through the interaction of a VM, which is a software emulation of a given OS, such as Linux Ubuntu or Windows 7 (Win7), and hardware devices, such as memory and clocks. AWS refers to their configured VMs as instances, which can be built from a given template or from scratch and then launched as a virtual server [16]. These VMs can do just about anything that a physical machine can do, such as run applications and browse the Internet. They can even be used for sandboxing, or isolating, malicious programs and scripts in order to investigate them while protecting the physical machine hosting the VM. However, since VMs have no physical parts, they must utilize the same hardware as the hosting machine and other co-located VMs. This includes items such as main memory, hard disk, and clocks [14], which provide excellent avenues to covertly extract information from a co-located VM.

## 2.2 Cloud Security

The cloud has many benefits over traditional computing, such as lower costs and better agility. We define agility as the ability to immediately scale up or down the size of cloud services, such as the number of VMs launched, or even shifting between cloud service models, such as moving from SaaS to IaaS. However, the cloud also introduces a number of security risks since physical servers are shared with other unknown, and ultimately untrusted, users [5]. With the cloud provider suppling the software, hardware, and infrastructure to run the necessary cloud services, the data itself is stored and accessed at a remote location (for example, AWS [16] defines these consolidated data centers as Availability Zones). In this respect, the cloud user does not have full control over the physical and logical security of their data, nor the objects that support their virtual computing. Software patches for the physical server's OS and/or hypervisor, maintained by the cloud provider, may not be up to date and data could be easily stolen and/or illegally sold, regardless of the cloud model selected [17]. Thus, hypervisor and VM security is a primary concern for cloud users. In this section, we discuss both the traditional security practices implemented by public cloud (using AWS as an example) and AWS's implementation of the Virtual Private Cloud (VPC).

### 2.2.1 Component Security

Every IT professional is trained on a multitude of methods on how to best secure their systems and software. These methods include software patching, security groups, network hardening, and physical security of rooms and devices. These methods are not just for private systems and networks, but also for cloud security since the cloud is, in its most fundamental form, a data center [18]. For example, the hypervisors must be patched routinely in order to eliminate discovered vulnerabilities that can be exploited by malicious users. Additionally, cloud administrators must create security groups that prevent non-privileged users from configuring and accessing the cloud hardware. However, while all cloud providers should be securing their systems in this manner, cloud security must also take additional measures into consideration, such as the access of VMs by their rightful owners.

In 2010, Durbano et al. [18] conducted a study on how to reliably secure a cloud environment, listing 20 security configuration recommendations. One of the case studies conducted on AWS found six of these recommendations were already implemented within its standard security model. In general, AWS implements multiple layers of authentication when interacting with VM instances, utilizes bastion hosts for user interaction, and wipes all data from the physical servers when users no longer require access to it.

### 2.2.2 Virtual Private Cloud

Amazon altered the face of cloud security by introducing the Virtual Private Cloud. By design, the VPC is not intended to strictly be a security feature of AWS, but rather a method to create a virtual network directly tied to a registered user's account [4]. In essence, this provides cloud users the capability to build and scale VM instances to easily mimic a physical network while logically isolating it, through separate subnets and MAC address space, from both the physical server network and other virtual networks in the same Region and Availability Zone. The massive size of the AWS infrastructure allows the separation of cloud services into 11 publicly accessible Regions that are independent of each other, allowing for high levels of stability and fault tolerance. Each AWS Region contains multiple Availability Zones that are independent of each other but allow logical VPC connections between them [16]. This enables more advanced networking techniques and options than the standard cloud model, such as giving users the opportunity to assign multiple public Internet Protocol (IP) addresses to a single instance and assigning persistent static private IP addresses for VMs that are repeatedly stopped and started.

Nevertheless, the VPC included some significant security features that were not previously available in the standard cloud model [4]. First, the VPC automatically generates a network-based Access Control List (ACL), allowing users to determine what general traffic and data is allowed to enter and leave their virtual network. Secondly, users have the capability to filter network traffic both to and from each instance in the VPC, creating a multitude of unique ACLs tailored to the specific use or function of the instance. In addition, security groups can be dynamically modified while instances are in use, instead of being forced to shut down and/or reboot instances. Lastly, users have the option of utilizing single-tenant hardware, or Dedicated Instances, where all instances belonging to a single VPC are initialized on physically isolated servers. While the security features provided by the VPC are only as effective as the knowledge and effort of the cloud user, the fact that implementation of the VPC has become mandatory for Amazon users [4] ultimately makes the AWS cloud more secure.

## 2.3 Co-Location Attacks and Detection

In this section, we discuss the rising trend of malicious cloud attacks known as co-location attacks. We begin by introducing the concept of the attack and why it is dangerous to cloud users. Next, we provide an overview of the basic implementation of the attack. Lastly, we discuss previous research conducted on confirming the presence of two co-located VMs, differentiating between detection methods feasible prior to the implementation of Amazon's VPC and those still viable after.

### 2.3.1 Co-Location Attacks

While public clouds inherited all of the "traditional" vulnerabilities, such as viruses, worms, and denial of service attacks originating from a source external of the cloud server, a new vulnerability emerged. By launching a VM instance on the same physical cloud server as a second instance, an adversary can now implement a co-location attack by either launching a denial of service attack originating from the same physical server or a side-channel attack. While each physical server runs a hypervisor application to create and control instance VMs, a denial of service attack can be conducted through a malicious VM exploiting vulnerabilities in the hypervisor, allowing the VM to overwork the computing constraints of the server and prevent any other instance located on that same server from functioning.

The side-channel attack is an extension of the traditional covert channel attack [3], which is using an open, unintended communications method to transmit data and information [19]. A simple example is an adversary planting a Trojan to access protected File A and transfer the data bit-by-bit through a coordinated effort with the adversary by locking (0 bit) and unlocking (1 bit) access to unprotected File B at set time intervals. As defined by Ristenpart *et al.* [3], the side-channel attack is the extraction of information across co-located VMs through the shared resources of the physical server. Examples of information that could be extracted are images, sensitive documents, password hashes, and cryptographic keys. Previous studies on this topic have identified multiple methods to collect the desired information. Ristenpart *et al.* [3] successfully demonstrated cross-VM information leakage by measuring computational loads on shared caches. Zhang *et al.* [11] preformed a Flush-Reload-based attack to count the number of items in a target's online shopping cart. Masti *et al.* [20] showed how this attack could be completed by measuring the temperature of the server's processors.

Co-location attacks pose great risks to all legitimate users of the cloud, especially organizations like the DOD that could potentially store sensitive files and data on cloud servers. Since cloud providers open their services to all users who are willing to pay, it is impossible to know who is sharing a server with whom. Co-location attacks are unpredictable and detection of an attack is usually known only after it has been conducted. This is because the data is passed via a patterned usage of hardware resources which is hard to detect early in the transmission process. Decreasing the likelihood of a successful attack requires complex patching and hardening of the physical server, hypervisor, and VM OS configurations [21]. While a cloud provider will allow the reservation and dedication of specific servers to help with both computing performance and security (AWS [16] calls them Dedicated Instances, for example) the large increase in cost is often too expensive for many DOD organizations. Instead, VM placement in today's cloud is based on algorithms with a number of factors such as VM instance type, time launched, number of servers in the cloud data center, and number of VMs in use [22].

### 2.3.2 Co-Location Methodology

A co-location attack has three primary phases. First, an adversary must have specific knowledge of the target VM. This knowledge includes but is not limited to the cloud provider, data center location, and IP address. Next, the adversary must launch one or more VM instances in the same data center as the target instances and determine if any one of its instances is co-located with one of the target VMs. If co-location is confirmed, then the attack can be implemented in the last phase; however, if an adversary can be prevented from accurately determining instance co-location, then the attack cannot be implemented effectively. In order to best combat this threat, it is best to counter its early attack phases. For this study, it is assumed that the adversary has done sufficient research in the first phase to begin attempting to co-locate its VM instances with the target.

### 2.3.3 Related Work

Using AWS as a point of reference, research on VM co-location detection falls into two primary time periods. We define these periods as Pre-VPC and Post-VPC, noting the dividing line as the point at which Amazon implemented the VPC as standard practice.

#### Pre-VPC

Previous work on co-location detection, as summarized in Table 2.1, has exposed a number of ways to easily exploit common network protocols and tools. Ristenpart *et al.* [3], the first to study the exploitation of the cloud with respect to co-location attacks, showed how it was possible to simply utilize packet response times and Domain Name Service (DNS) queries to determine the internal IP addresses of a cloud infrastructure and subsequently map it. Co-location determination then became a quick analysis of this data. Bates *et al.* [5] showed how an adversary can determine co-location by utilizing an active traffic analysis technique called watermarking, or the injection of a unique network flow signature, to fingerprint a target. This watermarking technique was conducted by controlling a target instance's network traffic through controlled packet delay to give it a uniquely identifiable pattern. If the target instance displays the same delay pattern as one of the adversary instances, then co-location has been identified. Herzberg *et al.* [6] determined that an adversary could deanonymize a target instance's private IP address and measure the hop-count of packet routing.

#### Post-VPC

Cloud providers have acted on the focused research of cloud security exploits and effectively countered the ability to utilize most of the pre-VPC detection techniques. For example, trace routes can no longer accurately determine the number of hops a packet travels inside the cloud infrastructure, private IP addresses are now dynamically allocated instead of statically assigned, and use of the VPC by Amazon EC2 has hidden private IP addresses from other cloud users as well as providing the means to allocate multiple private IP addresses for one VM. However, these security methods are not foolproof. Within the last year, Xu *et al.* [7] demonstrated that even with the introduction of the VPC, co-location of instances can still be determined by implementing latency-based network probing. Also, Varadarajan *et al.* [22] showed how an adversary can increase the probability of co-location in multiple cloud providers based on the time of day to launch instances, how long to delay launching instances after the target instance was launched, and the number of instances to launch. Taking past actions into account, it is safe to assume that these methods will soon cease to work for potential attackers. This now gives rise to an important question: As the cloud security teams quickly adapt and improve the defense of the cloud infrastructure, can an adversary still accurately determine co-location without raising any alarms?

Table 2.1: Summary of Co-location Detection Methods

| **Attack Method** | **Cloud Cartography** | **Watermarking** | **Topology Mapping** | **Latent Network Probing** | **Clock Skews** |
|---|---|---|---|---|---|
| **Researcher** | Ristenpart [3] | Bates [5] | Herzberg [6] | Xu [7] | Kohno [8] |
| **Year** | 2009 | 2012 | 2013 | 2015 | 2005 |
| **Resources Attacked** | Routing | Bandwidth, packet release | Routing | Routing, shared memory | System clock |
| **Tools Used** | nmap, hping, whois | PHP scripts | Hardware interrupts, whois, tracerouting | Tracerouting, memory locking sender and receiver, HTTPerf | CAIDA, TCP/ICMP requests |
| **Protocols Exploited** | TCP | TCP, UDP | SMTP, TCP, UDP, ICMP | HTTP, TCP | TCP, ICMP, NTP |
| **Counters** | - Dynamically assign Private IPs<br>- Obscure Traceroute info.<br>- Disable/obscure ping requests | - Dedicated path from VM to physical host<br>- VM underprovisioning<br>- Randomize outbound packet scheduling | - Block internal cloud VM communication<br>- Utilize firewall to limit internal communication | - More dynamic VM placement<br>- Randomize domain name generation<br>- Obscure traceroute paths | -Minimize clock skew |
| **Still Valid** | No | No | No | Yes | ??? |

## 2.4 Device Fingerprinting with Clock Skews

Clock skew is defined as the temporal deviation of a clock in a one-second period in reference to a control clock. In 2005, Kohno *et al.* [8] demonstrated that a physical computing device could be remotely fingerprinted using its clock skew. Those results confirmed the common belief that computing devices have unique clock skews, even among devices with seemingly identical hardware and software buildouts. Through the use of the simple network protocols TCP and ICMP, a technique was created utilizing packet timestamps to calculate a device's clock skew with respect to a given control device. Kohno *et al.* applied this technique to a controlled setting of five VMs in order to evaluate the differences between a real network and a virtualized network. Chen *et al.* [23] extended this work to fingerprint remote VMs in an effort to thwart a malware's ability to detect remote VMs. In 2015, Sheridan [14] derived VM clock skews from TCP timestamps to study the behavior of virtual OSs. To the best of our knowledge, no one has applied the clock skew fingerprinting technique to determine VM co-location in the cloud. We also extend the work of Kohno *et al.* by improving the method of estimating a clock skew, determining the number of timestamps required to provide a sufficient estimate, and studying the effect different network models have on clock skew estimation.

### 2.4.1 TCP Timestamps

The TCP was designed to be a reliable data transmission protocol, ensuring that clients received all intended data packets from the sender. In order to help optimize the efficiency of the protocol over paths with large bandwidths and very high data throughput speeds, the Timestamps option (TSopt) extension was added to the TCP packet header [24]. Increasing the overall packet size by an additional ten bytes, the TSopt provides two separate timestamps: the Timestamp Value (TSval), or timestamp at which a TCP packet is sent, and the Timestamp Echo Reply (TSecr), or the echo of the last TSval received.

Each timestamp is denoted as a four-byte integer that represents the number of clock ticks passed, most typically since system bootup [24]. The clock ticks reference a virtual clock that is proportional to the actual system clock of the device. The number of clock ticks per second passed in real time, or clock frequency, is predefined by the OS of the system and ranges from 1-1000 Hz. For example, the TCP clock frequency in Windows machines is 10 Hz while Linux machines can be 100 or 250 Hz [23].

In order to enable TSopt, both communicating parties (client and server) must agree to apply it during the initial TCP handshake with the client including the TSopt in the initial SYN packet [24]. If the option is disabled by just one party, then no timestamps are included in the TCP options header. While some OSs enable TSopt by default to help improve TCP efficiency, such as Linux, other OSs, such as Windows, disable the option by default [25]. Kohno *et al.* [8] demonstrates some methods that can "trick" machines into enabling the TSopt even if it is disabled by one party. For our study, we assume that an attacker has implemented some forced TSopt enablement method or the TSopt is not disabled as part of the target's configuration, thus all hardware devices and VMs have the TSopt enabled.

### 2.4.2 Estimating Clock Skew

The estimation of clock skews from a remote vantage point requires the transformation of TCP timestamps ($T$), measured in units of clock ticks, to relative clock offset values ($y$), measured in units of seconds, and then fit to a linear regression model. This is done through a series of calculations introduced by Kohno *et al.* [8] and expanded on by Sheridan [14]. It is important to note that we define this process as an estimation vice a calculation since the true clock skew is generally not known and various network delays introduce enough variance in the clock offset values to prohibit a true calculation by this method. For this reason, the linear regression of the transformed data points provides an estimate of the general clock skew trend. Table 2.2 lists the components required to estimate a clock skew as well as their definitions.

Table 2.2: Clock Skew Variables

| Variable | Definition | Unit of Measure |
|---|---|---|
| $T$ | TCP Timestamp of target VM | ticks |
| $t$ | Timestamp when the host system received TCP packet | seconds |
| $f$ | Frequency of target VM's virtual clock | Hz |
| $x$ | Elapsed host system time from first packet received | seconds |
| $v$ | Elapsed number of target VM clock ticks since first packet sent | ticks |
| $w$ | VM timestamps adjusted to account for OS frequency | seconds |
| $y$ | VM clock offset with respect to elapsed host time | seconds |

After capturing a number of $n$ packets from a specific source, we define the set of TCP timestamps as $T = \{T_1, T_2, T_3, ..., T_n\}$ and the set of packet receipt timestamps as $t = \{t_1, t_2, t_3, ..., t_n\}$. Since $T$ is measured in units of clock ticks, we must transform these values into units of seconds in order to properly compare them to $t$. We do this by determining the frequency of the target VM's virtual clock as shown in Equation (2.1). With the frequency of the target's virtual clock pre-defined by its OS, we cannot assume to know what the OS or frequency is and must derive it by taking the ratio of elapsed clock ticks from the first to last packet received to the total elapsed time of packet collection. Due to the number of delays that may influence packets in transit, we round the calculated value to the closest frequency known to be common. For example, calculated frequencies of 247.3 Hz or 261.1 Hz would round to 250 Hz. The newly rounded frequency value is then used to transform each individual TCP timestamp. We can also determine the granularity of the TCP timestamp directly from the frequency measurement of the target's virtual clock. For example, a TCP timestamp from an OS with a virtual clock of 250 Hz would have a precision granularity of $\frac{1}{250} seconds$. It is important to note that the timestamp recorded when a packet is received is generally detailed to the microsecond. This difference in granularity between the two sets of timestamps ultimately induces a truncation error in these equations and graphically displays as a band of data points instead of a line, as seen later in Figure 3.3.

**Virtual Clock Frequency**

$$f = \frac{T_{Last} - T_1}{t_{Last} - t_1} \tag{2.1}$$

In order to normalize our data to make our graphical results easier to understand, we define $i$ as the $i$-th packet received and use Equation (2.2) to show a relative timelapse, in seconds, from the first packet received. Similarly, Equation (2.3) shows the relative timelapse, in clock ticks, from the first TSval sent. We use Equation (2.4) to correct the normalized $v_i$ into units of seconds. Equation (2.5) determines the specific clock offset in relation to the host system time. A linear regression model, defined in Equation (2.6), is then fit to the offset values with $\hat{y}_i$ as the predicted clock offset value, $b$ as the y-intercept, and the slope $a$ reflecting the estimated clock skew.

**Normalized Host Time**

$$x_i = t_i - t_1 \tag{2.2}$$

**Normalized Target Time**

$$v_i = T_i - T_1 \tag{2.3}$$

### Adjusted Target Time

$$w_i = \frac{v_i}{f} \tag{2.4}$$

### Clock Offset

$$y_i = w_i - x_i \tag{2.5}$$

### Linear Regression Model

$$\hat{y}_i = ax_i + b \tag{2.6}$$

# CHAPTER 3:
# Methodology and Analysis

In this chapter, we look to derive our methodology for conducting live experiments in the AWS GovCloud, which will be described in Chapter 4. We begin by introducing a simple two-node network on which we conduct our initial analysis of test configurations, system clock versus TSval behaviors, and the Ordinary Least Squares (OLS) estimator. Next, we introduce four other estimators and analyze their performance. We then extend our findings into a controlled server-cluster in order to determine the validity of our test configurations in a setting more representative of a cloud environment. Lastly, we analyze our estimator methods in a simulator against a known clock skew to determine how they perform under various network traffic models and optimize the Wave Rider Estimator to account for an active-collection approach.

## 3.1 Single-Server Experiment

This section introduces our initial methods to understanding clock skew estimation and how to best collect and analyze TCP timestamps. We begin by discussing the configuration of a simple single-server test network and various testing configurations regarding target and data collection platforms and the number of packets collected in a trial run. Next, we discuss the differences and influences that TCP TSvals have on skew estimation over simple OS system call timestamps. Lastly, we discuss the results from our analysis of testing configurations, as well as OLS as a clock skew estimator.

### 3.1.1 Setup and Configuration

Our investigation starts by creating a controlled testing environment in order to simulate the basic functionality of a cloud architecture, demonstrate the basic implementation of TCP timestamp collection, and analyze the clock skew estimation technique described in Section 2.4.2. With Linux Ubuntu 14.04 LTS OS as a base image, we built multiple VMs on two laptop computers: an Apple MacBook Pro (Intel Core i7 CPU @ 2.50 GHz, 16 GB RAM) and a Dell Inspiron 15 Windows 10 (Win10) laptop (Intel Core i3 CPU @ 1.90 GHz, 8 GB RAM). The hypervisor chosen to host the VMs was Oracle VirtualBox v5.0.20, primarily due to the benefit of integrating Vagrant, an automated VM building tool, to launch instances on both laptops as well as in the AWS cloud for future live tests. A bridged network connection to the Naval Postgraduate School (NPS) local intranet was configured for all VMs to both induce typical network latency affects on TCP traffic as well as better simulate normal cloud routing behavior. Figure 3.1 shows the physical network configuration for all of our initial experiments. For simplicity, we use the term *Data Collector* for any VM or native OS defined as the base reference on which to compare another VM's clock drift and we call any VM whose skew we are estimating a *Target*. In addition, all VMs and native laptop OSs had Python v3.5.1 (a programming language) installed in order to run a TCP traffic-generating script. While many OSs today disable the TCP TSopt by default, we assume that the attacker can execute some method to force the target to enable this option during the TCP three-way handshake, such as the technique implemented by Kohno *et al.* [8]. Since timestamps can only be sent when both parties have TSopt enabled, we configured all VMs and host machines with this feature enabled [24].

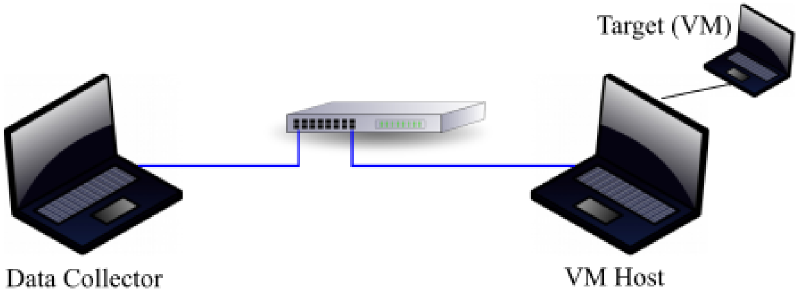


Figure 3.1: Network Configuration for Initial Experiments

In order to conduct our experiments, we wrote a Python script to generate TCP traffic between Target and Data Collector VMs, parse TCP packet capture files, estimate clock skews, and conduct statistical analysis on the skew estimation method. To generate our traffic, we wrote simple cooperative client-server scripts where the server, executed from the Target VM, would periodically generate packets and send them to the Data Collector (client), executed from a separate laptop on either the native OS or a hosted VM, upon establishing a connection. Early versions of this script had the server execute a system call to its OS to collect the current timestamp and wrap it in a TCP packet to the client.

We learned early on that a one-second delay between system timestamp calls was necessary in order to accommodate unanticipated networking delays that would result in two packets arriving almost simultaneously. Without this delay, Python would attempt to read the timestamps in these two packets as a single value and generate an error, resulting in a failed test run. The intent behind using the system timestamp calls was to simulate the TCP timestamps that would normally be generated by TSopt-enabled traffic until a method was created to correctly parse the timestamp values from the TCP packets. We also tested delay intervals of 0.25, 0.5, and 2.0 seconds, respectively. We found the intervals of 0.25, 0.5, and 1.0 seconds prevented almost all of the collision issues, though some did occur on occasion. The rate of collisions generally decreased as the delay time increased. We observed no collisions in our tests with the 2.0 seconds time delay; however, we decided to use the 1.0 second delay due to the low frequency of collisions, the overall packet collection time being lower, the collection time in seconds being roughly equal to the number of packet samples collected, and an anticipatory method of an attacker intentionally limiting the rate of VM probing in order to evade detection. The implementation of packet capturing and a parsing script rendered system timestamp calls obsolete, but the 1.0 second packet delay was left in the script in order to keep the total collection time easily estimated and maintain the assumption that the adversary is trying to evade detection.

Various configuration combinations of VMs and native laptop OSs playing the roles of Data Collector and Target, as well as the number of packets collected, were tested in order to determine an optimal number of timestamps to collect, as well as to observe any noticeable differences, if any, in the collected timestamp patterns. The various configurations tested are listed in Table 3.1. In order to automate clock skew estimation and analysis, we wrote a Python script that would read in the captured timestamp values and generate both a graphical representation of the data and the skew estimator, as well as a statistical analysis of each clock skew estimator. Metrics recorded by the script for each estimator consisted of estimated skew, Coefficient of Correlation (Equation 3.1), Coefficient of Determination (Equation 3.2), Mean Absolute Deviation (Equation 3.3), and Sum of Squares for Error (Equation 3.4) [26].

Table 3.1: Initial Test Configurations

| Configuration No. | Data Collector Platform | Target Platform | No. of Samples Collected | Used Real TCP TSval |
|---|---|---|---|---|
| 1 | Windows | Mac - Ubuntu | 10 | No |
| 2 | Windows | Mac - Ubuntu | 150 | No |
| 3 | Windows | Mac - Ubuntu | 300 | No |
| 4 | Windows | Mac - Ubuntu | 500 | No |
| 5 | Windows | Mac - Ubuntu | 600 | Yes |
| 6 | Windows | Mac - Ubuntu | 1000 | No |
| 7 | Mac | Windows - Ubuntu | 300 | No |
| 8 | Mac | Windows - Ubuntu | 600 | Yes |
| 9 | Mac | Windows - Ubuntu | 1000 | No |

**Coefficient of Correlation**

$$r = \frac{s_{xy}}{s_x s_y} \tag{3.1}$$

**Coefficient of Determination**

$$R^2 = \frac{s_{xy}^2}{s_x^2 s_y^2} \tag{3.2}$$

**Mean Absolute Deviation**

$$MAD = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n} \tag{3.3}$$

**Sum of Squares for Error**

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{3.4}$$

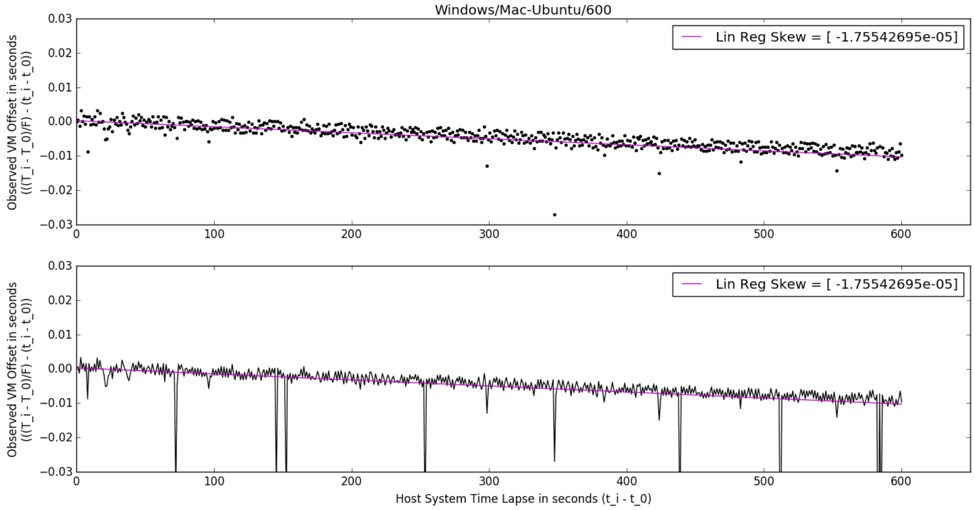
### 3.1.2 Effect of TCP Timestamp Value Resolution

While the initial skew models were conducted by processing timestamps generated by system calls by the traffic generation scripts on both the Data Collector and Target, we knew our experimentation in the AWS cloud would require the collection of actual TCP timestamps in order to hold validity in a real-world setting. Running Wireshark on the Data Collector, a program commonly used to collect network traffic and conduct packet analysis, we were able to successfully collect the TCP packets sent from the Target. We extracted the TCP TSvals from each packet's header as well as the system timestamps marking the arrival of each data packet at the Data Collector. These values were then analyzed with the same Python script as the system-called timestamps.

The resultant output, shown in Figure 3.2, depicts the captured data points as both individual dots and as a line. While the top plot shows the expected band of data points (with a few visually identified outliers), the lower graph illustrates a periodicity or wave-like pattern of the plot values as the line travels up and down following a general linear downward slope. Taking the first 20 packets of the packet capture file, we processed each timestamp pair individually, noting the overall data trend and tabulated the output. The results of these calculations are shown in Table 3.2 with the first seven values in the last column showing the periodicity trend. The second clock offset value ($y$) starts at 0.002707 and then progresses with smaller values, reaching -0.000900 before jumping back up to 0.001900. This pattern is repeated in all 20 packets collected, Figure 3.2 shows the pattern throughout the 600 collected packets.

It is important to note the number type used for the two timestamps. The system timestamp is represented by a double-precision floating decimal and accurate to the microsecond, or $10^{-6}$, whereas the TCP TSval is a 32-bit integer that represents the number of clock ticks of a device's virtual clock [24]. By using Equation (2.1), we find that this particular VM's clock ticks ($f$) are accurate to $1/250$ of a second, or four milliseconds. In other words, by comparing a less granular form of measurement to a higher granularity form, truncation errors are induced in the VM offset value by transforming TSvals from units of ticks to seconds, leading to the graphical periodicity.

We argue that the differences in granularity between the two timestamp values should not negatively impact our estimate of clock skews. Since the determination of the clock skew



This figure depicts two plots of the same data points. The top plot displays the data points as dots, the bottom plot as a line. Both plots estimate the clock skew with the OLS estimator. The x-axis in both plots represents the elapsed time of the TCP packet collection as referenced by the Data Collector. The y-axis represents the clock offset $y$ of the Target with respect to the Data Collector. The slope of the estimator model $\hat{y}$ is the clock skew.

Figure 3.2: Pcap Initial Test–Configuration No. 5

from the clock offset values is an estimate and not a calculation, the general sloping trend of the data points is sufficient to generate a reliable clock skew value. While it would be ideal that the data points represent a line of data instead of a band, as this would drastically help in the visual identification of outlier data points and skew estimation, a tightly grouped band of data points generated by a relatively small periodicity range (i.e., the difference of values between the band's upper and lower threshold limits) should be just as helpful.

### 3.1.3 Results

We began our analysis of the initial round of tests by focusing on the number of timestamps that would provide us with sufficient information to accurately estimate clock skews. The reasoning behind this was to discover a collection size that would generate fairly consistent results while not incurring excessively long test periods. With the Python server script written to send a TCP packet every second, a collection size equates to the number of seconds

Table 3.2: Packet Periodicity

| Timestamp | TSval | f | x | v | $\Delta v$ | w | y |
|---|---|---|---|---|---|---|---|
| 1463519524.060397 | 40853 | 250.6766064 | 1.001293 | 251 | 251 | 1.004 | 0.002707 |
| 1463519525.061631 | 41103 | 250.1835056 | 2.002527 | 501 | 250 | 2.004 | 0.001473 |
| 1463519526.062796 | 41353 | 250.0258012 | 3.003692 | 751 | 250 | 3.004 | 0.000308 |
| 1463519527.064004 | 41603 | 249.9438202 | 4.004900 | 1001 | 250 | 4.004 | -0.000900 |
| 1463519528.065204 | 41854 | 250.0948754 | 5.006100 | 1252 | 251 | 5.008 | 0.001900 |
| 1463519529.066423 | 42104 | 250.0282919 | 6.007319 | 1502 | 250 | 6.008 | 0.000681 |
| 1463519530.067636 | 42354 | 249.9810895 | 7.008532 | 1752 | 250 | 7.008 | -0.000532 |
| 1463519531.068857 | 42605 | 250.0702229 | 8.009753 | 2003 | 251 | 8.012 | 0.002247 |
| 1463519532.070123 | 42855 | 250.0271838 | 9.011019 | 2253 | 250 | 9.012 | 0.000981 |
| 1463519533.071344 | 43105 | 249.9940031 | 10.012240 | 2503 | 250 | 10.012 | -0.000240 |
| 1463519534.072525 | 43356 | 250.0585625 | 11.013421 | 2754 | 251 | 11.016 | 0.002579 |
| 1463519535.073701 | 43606 | 250.0291304 | 12.014597 | 3004 | 250 | 12.016 | 0.001403 |
| 1463519536.074888 | 43856 | 250.0042262 | 13.015784 | 3254 | 250 | 13.016 | 0.000216 |
| 1463519537.076173 | 44107 | 250.0522567 | 14.017069 | 3505 | 251 | 14.020 | 0.002931 |
| 1463519538.077328 | 44357 | 250.0296276 | 15.018224 | 3755 | 250 | 15.020 | 0.001776 |
| 1463519539.078542 | 44607 | 250.0087367 | 16.019438 | 4005 | 250 | 16.020 | 0.000562 |
| 1463519540.079705 | 44857 | 249.9911860 | 17.020601 | 4255 | 250 | 17.020 | -0.000601 |
| 1463519541.080875 | 45108 | 250.0309348 | 18.021771 | 4506 | 251 | 18.024 | 0.002229 |
| 1463519542.082124 | 45358 | 250.0128788 | 19.023020 | 4756 | 250 | 19.024 | 0.000980 |

Note: The column of frequency values is not a fixed value as in Equation (2.1) but relative to each packet. It was calculated to show how the virtual clock frequency is fairly constant and that any two packets can be used to determine the fixed value. Also, the $\Delta v$ column is used as a reference to demonstrate the truncation error as time in decimal form is truncated to an integer value. Lastly, the first packet was removed from the table as it is the baseline reference for the other 19 packets, thereby having a value of 0 for each non-timestamp column.

the test lasts. For example, a test run collecting 100 packets will take approximately 100 seconds to complete. With the configuration of the Windows laptop as the Data Collector and the Ubuntu VM hosted on the MacBook as the Target, multiple trial runs were conducted on collection sizes of 10, 150, 300, 500, 600, and 1000 packets. We observed that collecting fewer than 300 packets resulted in clock skew estimations with a larger standard deviation, as compared to collection sizes greater than 300 packets, within each set of trial runs. The standard deviation values generally decreased as the collection size increased. As shown in Table 3.3, for collection sizes above 500 packets the skew standard deviation of each test set did not decrease with the increase in sample size. We decided that further testing would use a collection size of 600 TCP packets as this would provide us both consistent results and an even ten minutes to collect packets on each VM.

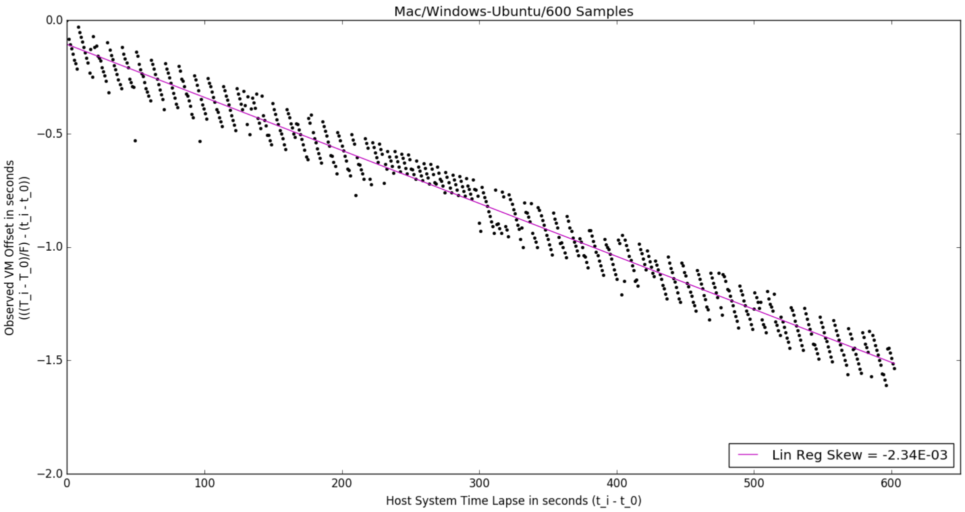
Table 3.3: The Effect of Collection Size on Skew Estimation

| Collection Size | 10 | 150 | 300 | 500 | 600 | 1000 |
|---|---|---|---|---|---|---|
| Trial Run 1 | 4.48E-05 | -1.15E-06 | -1.27E-07 | 1.61E-07 | 1.45E-07 | 2.66E-07 |
| Trial Run 2 | -1.01E-05 | 4.88E-07 | 1.22E-06 | 5.03E-07 | 1.59E-07 | 1.67E-07 |
| Trial Run 3 | -3.03E-05 | -5.96E-07 | 3.71E-07 | 4.99E-07 | 1.95E-07 | 9.34E-08 |
| Std. Deviation | 3.89E-05 | 8.33E-07 | 6.81E-07 | 1.96E-07 | 2.58E-08 | 8.66E-08 |

We then looked at various OS configurations for the Target and Data Collector roles to determine if there would be any significant difference in the results. Using a sample size of 600 packets, we conducted multiple trial runs in each Target/Data Collector configuration. We noted the data points were more periodic and in a more defined band when the Mac laptop was acting as the Data Collector and the Windows laptop hosted the Target, as depicted in Figure 3.3 with Configuration No. 8. All generated graphs are configured with the x-axis representing time $t_i$, in units of seconds, as referenced by the Data Collector. The y-axis represents the Target's clock offset value $y_i$, in units of seconds, in respect to the Data Collector. The data points form a visual band with a noticeable sloping trend. It is the slope of the linear regression model fit to these data points that estimates the Target's clock skew.

In addition, the skew estimates for Configuration Nos. 7-9 were generally larger when compared to the estimates of Configuration Nos. 4-6, as illustrated in Figure 3.4. However, while the graphical representations of the clock offset values are vastly different, the overall behavior of the skew estimator did not change between the various OS configurations in that the estimations generated were visually verified as following the sloping trend of the data. We thus concluded the differences in these results could be attributed to the computing behavior of the OSs themselves in how they handle both VMs and time keeping. Since graphical output between OS configurations can be vastly different, the Data Collector's OS must remain constant for all trial runs in a test set in order for the resulting skew estimation to have any value since clock offset values are relative to the Data Collector itself.

Lastly, we analyzed the regression model for estimating the clock skews. Following Sheridan's approach [14] of estimating clock skews with a Simple Linear Regression model, we derived an estimate by leveraging the linear modeling class methods within Python's



This figure depicts the skew estimation results using the OLS estimator. The test was conducted with Configuration No. 8.

Figure 3.3: OLS Estimator–Configuration No. 8: Mac/Windows-Ubuntu/600 Samples

Scikit-Learn v0.17.1 module in our clock skew analysis script. The Simple Linear Regression model, also known as OLS, is the most common and simplest approach to determining a linear regression fit and it provided us a baseline on which to compare alternate skew estimators. This regression model is generated by fitting a line through the data points such that the sum of squared residuals, or the total distance between the predicted and observed data points, is as small as possible [27].

### 3.1.4 Discussion

Overall, our observations revealed the OLS model to not be reliable as an estimator. The values for the coefficients of correlation $r$ (Equation 3.1) and determination $R^2$ (Equation 3.2) recorded by our statistical analysis routine revealed an overall weak fit of the OLS regression model to the data points. With a maximum value of 1.0 signifying the variance in the dependent variable as completely explained by the independent variable and a perfect fit of the data points to a regression line, a “good fit” is generally considered to have coefficient values closer to 1 than 0 [26]. However, the coefficients of correlation and determination

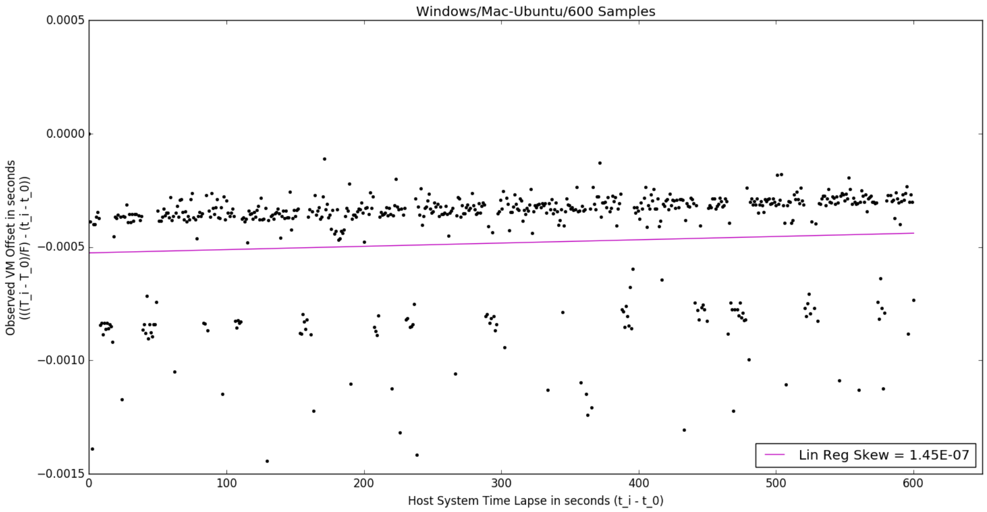


Figure 3.4: OLS Estimator–Configuration No. 5: Windows/Mac-Ubuntu/600 Samples

for the OLS model were generally found to support a weak fit to the data. A test set of four trial runs returned an average $r$-value of 0.4097 and an average $R^2$-value of 0.1834, as shown in Table 3.4. These low coefficient values imply that clock skew is not the driving factor for the patterns exhibited in the data, but rather some additional factors (most likely random network delays) are a larger influence than we initially anticipated.

Table 3.4: OLS Statistical Metrics

| Trial No. | Skew | $r$ | $R^2$ |
|---|---|---|---|
| 1 | 3.54E-05 | 0.4922 | 0.2422 |
| 2 | 3.67E-05 | 0.5613 | 0.3150 |
| 3 | 2.96E-05 | 0.2424 | 0.0588 |
| 4 | 6.66E-05 | 0.3430 | 0.1177 |
| Average | 4.21E-05 | 0.4097 | 0.1834 |

Additionally, the regression model was found to be highly influenced by outlier data, which are caused by random delays experienced by a TCP packet in transit. The most common example is a queuing delay built up at routers and switches during high traffic periods. The delay values fluctuate as general packet sizes and flow volume constantly increase and decrease. Another example is random processing delay from the host machine sending the packet. VMs work through the management of a hypervisor, which is an application on the host OS and must share processing resources with other applications. This in turn could force a packet to wait longer than normal to be sent from the machine, adding extra time to the overall trip. We found that while increasing the number of timestamps collected helped to average out outlier data points and normalize variance, a large enough outlier or a concentration of outliers will still affect the skew estimation, as shown in Figure 3.5. This in turn could potentially provide a false positive or false negative outcome when comparing two clock skew estimates. With this in mind, we determined that more robust methods of linear regression were needed in order to provide a more reliable clock skew estimate.

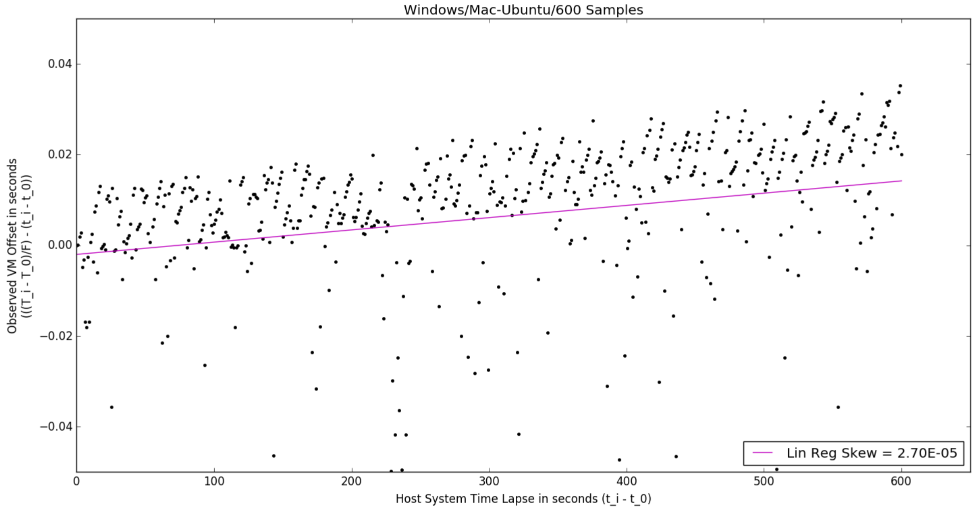


Figure 3.5: A Skew Estimate with OLS that is Biased Toward an Outlier Concentration–Configuration No. 5

## 3.2 Searching for Better Regression Methods

Our results from the OLS model drove us to consider additional linear regression methods for estimating the clock skew. In particular, we wanted to look at more robust regression methods, or methods that are not as influenced by outlier data points. We first evaluated two established robust regression methods, the Theil-Sen and Random Sample Consensus (RANSAC) linear regression methods [27]. We also looked at a method that would be resistant to outliers and extreme data variance by smoothing, or averaging, the data points using a moving average [26]. Our last method analyzed is one we created that fits only the data points with minimum delay times and is not influenced by outlier data, called the Wave

Rider model. We evaluate each method on the same physical network as the OLS method and analyze the test results to determine what method returns the most consistent, reliable skew estimation.

### 3.2.1 Theil-Sen

The first alternate estimator tested was the Theil-Sen Regression. It is considered a more robust method of determining a linear regression line over OLS due to the algorithm's design of taking the median of all slopes $s$ as determined from a sample of two-pair data points (i.e., $(y_j - y_i)/(x_j - x_i)$) from the collected data set [28]. In essence, it is supposed to be more resistant to outlier data points and becomes useful in multivariate data analysis. Theil-Sen Regression is also designed to not make any assumptions about the statistical distribution of the data, adding to its robustness against outliers [27]. We derived the Theil-Sen Regression skew estimate by using Python's scikit-learn 0.17.1 module in our Python skew analysis script.

The Theil-Sen estimator showed overall improvement on estimating skews over the OLS estimator. In general, the estimator performed with more resilience against outlier data points, successfully demonstrating its robustness. As depicted in Table 3.5, applying the Theil-Sen estimator to the same data points analyzed in Table 3.4 resulted in slightly improved $r$ and $R^2$ values, 0.4133 and 0.1967 respectively, over the OLS estimator.

Table 3.5: Theil-Sen Statistical Metrics

| Trial No. | Skew | $r$ | $R^2$ |
|---|---|---|---|
| 1 | 2.74E-05 | 0.3807 | 0.1449 |
| 2 | 4.48E-05 | 0.6843 | 0.4683 |
| 3 | 3.35E-05 | 0.2737 | 0.0749 |
| 4 | 6.11E-05 | 0.3143 | 0.0988 |
| Average | 4.17E-05 | 0.4133 | 0.1967 |

There was one large disadvantage to this estimator, however. Due to the nature of the estimator's algorithm design, the selection of the median slope from a random subset of slopes causes the estimated skew to rarely repeat when recalculated. Essentially, this greatly decreases the reliability of the model as an estimated skew could be generated that is completely wrong. Figure 3.6 illustrates such an example as the OLS estimator generates a skew that trends positive with the data points while the Theil-Sen estimator generates a skew that is grossly negative. In the end, while this estimator showed general promise, the lack of repeatable estimates led us to look for a better estimator.

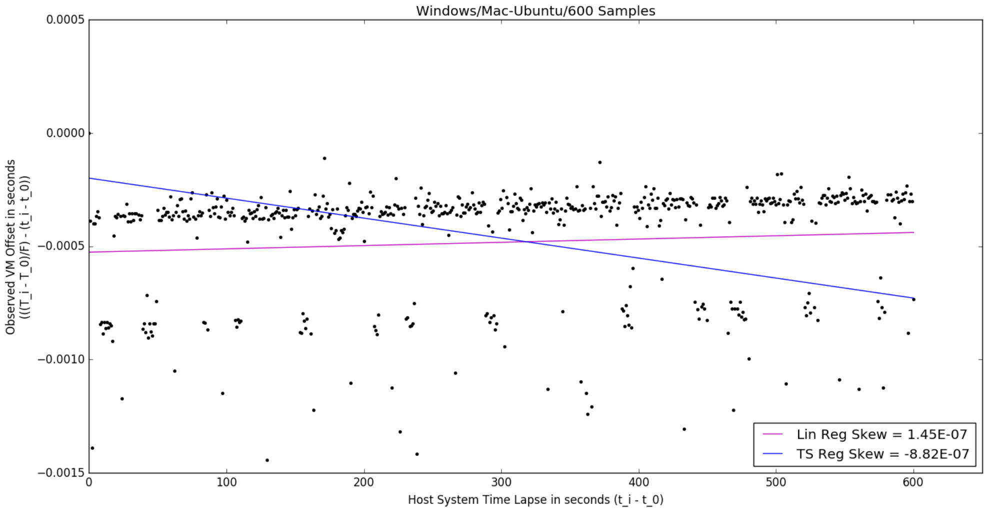


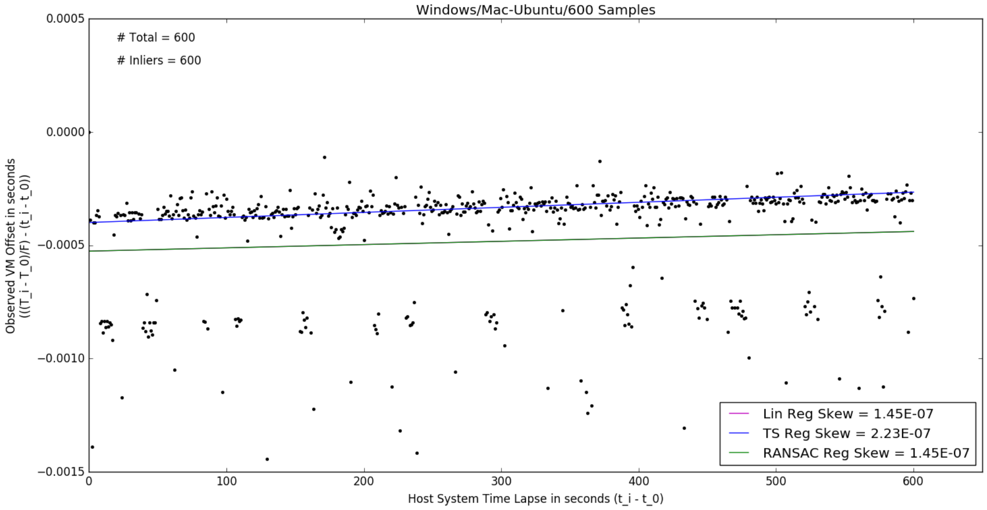
Figure 3.6: Theil-Sen Estimator–Configuration No. 5: Windows/Mac-Ubuntu/600 Samples

### 3.2.2 RANSAC

RANSAC Regression is another robust method of fitting a linear regression. The basic algorithm samples a defined number of data points and fits a linear model to them. The rest of the data points are then classified as either inliers or outliers as determined by being within a customizable parameter called residual threshold, or the maximum "vertical" distance allowable between the actual data point and the predicted data point [27]. The number of inliers is recorded and the steps are repeated either a defined number of times or when the number of inliers reaches another predefined threshold. A regression line is returned that fits a new model to the set consisting of the maximum number of inlier data points [29]. It is the ability to factor out and ignore the outlier data that defines this method as robust. We derived the RANSAC skew estimate by using Python's scikit-learn 0.17.1 module in our Python skew analysis script using the default algorithm settings with the exception of the residual threshold value.

Results from the RANSAC estimator showed better performance than both the OLS and Theil-Sen estimators. The *r* and $R^2$ values were overall higher, generally ranging from 0.3

to 0.5 with some values as high as 0.77. Unfortunately, these results hinged on the residual threshold value of the RANSAC algorithm. Figure 3.7 illustrates the RANSAC estimator with a residual threshold value of 0.01. Since the variance of the clock offset values is small, all 600 data points are considered to be inliers as determined by the algorithm. The end result of this estimation is a skew estimate that equals the OLS model. We argue that this particular model is in error as we can visually define a number of data points as potential outliers.



Note: The green RANSAC regression overlays and hides the red OLS regression.

Figure 3.7: RANSAC Estimator/0.01 Residual Threshold–Configuration No. 5: Windows/Mac-Ubuntu/600 Samples

Taking into account the range of clock offset values, we reestimate the skew by adjusting the residual threshold value to 0.0002. The results of this estimation, as shown in Figure 3.8, are noticeably different. Besides a different skew estimation value, we see a number of data points identified as outliers, annotated as red plot points, as well as the regression line shifted up into the grouping of inlier data points. While this method shows great promise as an accurate and reliable estimator, we argue that the deliberate manipulation of the residual threshold value to ensure the inclusion of all inliers and the exclusion of all outliers is an inconvenient flaw. Without knowing exactly how the clock offset values will range, the residual threshold value will have to be continually adjusted in order to get the most reliable estimation. The time necessary to do this on a large scale, such as in a cloud environment (i.e., 30 - 50 VMs), makes RANSAC a highly inefficient estimator and leads us to search for a more efficient method.

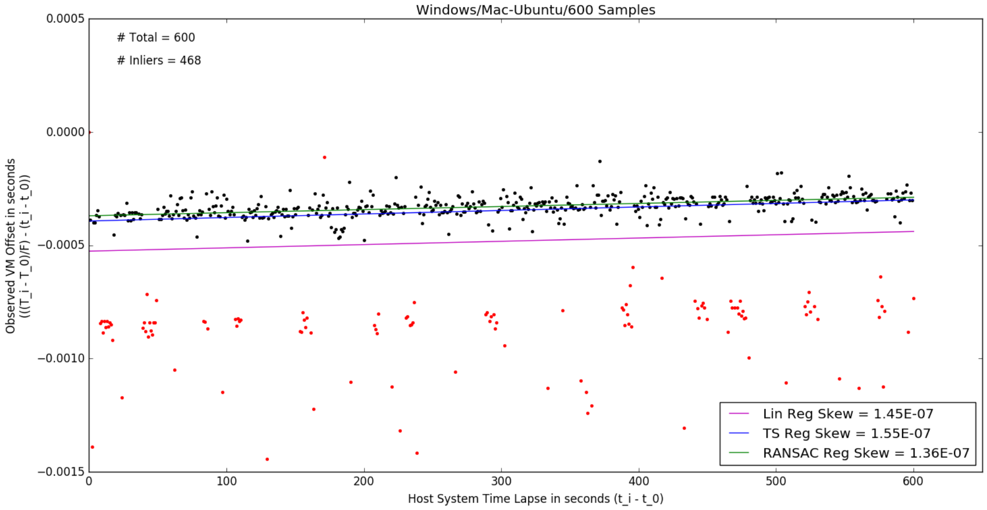


Figure 3.8: RANSAC Estimator/0.0002 Residual Threshold–Configuration No. 5: Windows/MAc-Ubuntu/600 Samples

### 3.2.3 Moving Average

Using the concept of moving averages, which were designed for time series data such as quarterly or multi-year trends, we created new data points from averaging a sliding window of a determined number of consecutive data points [26]. For example, a sliding window of three data points on data set $A$ creates the first moving average point by averaging $A_1$, $A_2$, and $A_3$, the second point is created by averaging $A_2$, $A_3$, and $A_4$, and so on. The advantage of this method is the ability to "smooth" the data, reducing any potential variance and influence by outliers in the data in order to provide a robust regression with higher correlation values. Our algorithm generates a new data set based on moving averages and then derives a skew estimate with the Simple Linear Regression module in Python's scikit-learn. We initially tested sliding window sizes of 3, 5, and 7 data points to determine what window size worked best. Finding little difference from OLS in the initial results, we increased the window size to 50 data points.

Our results gathered for the Moving Average estimator showed skew estimates that were relatively close in value to those generated by the OLS estimator. This was not surprising as the data points were averaged from the original data set and the final skew estimation derived with the OLS estimator. Unlike OLS, however, the Moving Average estimator has much stronger statistical support on the smoothed data with $r$ and $R^2$ values averaging 0.98; Figure 3.9 illustrates why this is so. As the lower graph depicts the data points as a black line, the three- and 50-point Moving Average data sets (red and blue lines, respectively) show a definitive smoothing of the data variance, with the variances decreasing as the window size increases. Averaging the data points ultimately eliminates most of the outliers, producing a more reliable estimate.

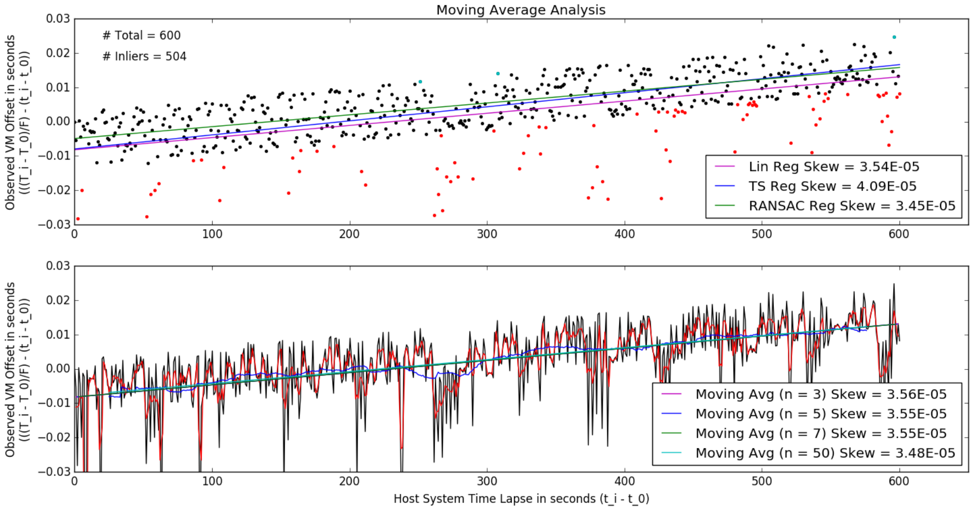


Figure 3.9: Moving Average Estimator–Configuration No. 5: Windows/Mac-Ubuntu/600 Samples

Regarding sizes, larger is better. However, since the number of data points lost to averaging with this method is equal to one less of the window size (i.e., a total of 49 data points are lost with a window size of 50), too large of a window could degrade the performance of the estimator as there are too few data points to analyze. We chose to focus our tests specifically on the 50 data point window size since our data set consisted of 600 packets. This gave us 551 data points in the moving average subset, which Section 3.1 showed is enough data points to derive consistent, reliable skew estimates.

Overall, the Moving Average estimator showed promise. However, while most of the outliers are averaged out with this method, a relatively large outlier or a large concentration of outliers at either end of the data set would have a strong influence on the skew estimation. This drove us to look for a method that was more resilient against any outlier.

### 3.2.4 Wave Rider

The Wave Rider estimator is a technique we created after investigating the behavior of plotted TSval offsets. As mentioned in Section 3.1, the natural periodicity of the offset values due to the granularity difference between the TSvals and system clocks generates an established band of data points that follows a general trend (i.e., the clock skew). We noticed that each band created a natural upper threshold boundary since all true outliers were always well below the band of data points. Analyzing our timestamp transformation equations more closely, we realized that the upper-bound data points represented minimal delay values, or points whose offset value more closely represented true clock drift. Outliers, on the other hand, are caused by unusually large network delays. We theorized that by picking at least two points with minimum delay values would provide an estimated skew value that would better resemble the true clock skew and would never be influenced by outliers.

With this in mind, we created an algorithm that recursively selects points that are "higher" than the data points before and after it, returning a dataset typically between two and ten points. A skew estimate is then derived using the Simple Linear Regression module in Python's scikit-learn library, leaving a regression line that rides on top of the band of data points. This is graphically shown only as long as the distance from the first point in the new dataset to the last point.

Our initial results with Wave Rider were very promising. As shown in Figure 3.10, our estimator successfully followed the slope of the data points and was not influenced by any of the "true" outlier data points, colored red as defined by RANSAC. In this figure, we argue that RANSAC incorrectly identified data points within the upper-bound region as outliers since those points represent packets with minimum delay values, an error derived from RANSAC's residual threshold parameter. As such, it is these two factors which cause Wave Rider to outperform the other four estimators. The ability to ignore outlier data points makes this estimator the most robust and allows it to operate on any traffic environment, regardless of amount of variation in the data points. Additionally, being able to derive a skew estimate that more closely reflects a device's true skew should ultimately allow for a more accurate comparison between two VMs. Statistically, the $r$ and $R^2$ values were generally the greatest compared to the other four estimators.

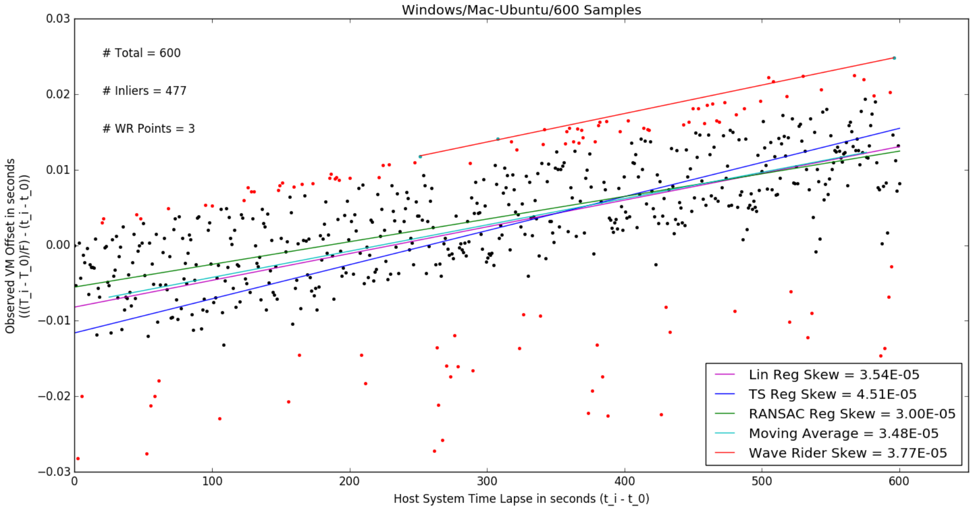


Figure 3.10: Wave Rider Estimator–Configuration No. 5: Windows/Mac-Ubuntu/600 Samples

## 3.3 Type I Hypervisor Experiment

While the initial single-server experiments were productive for testing the Python scripts and analyzing the various clock skew estimators, we needed a test network that could more realistically simulate a public cloud environment. Using a multi-server environment provided us with a way to see skew estimates from two VMs that were co-located and two that were on separate physical servers. Utilizing a Type I hypervisor cluster that is more closely representative of the typical VM data center, we were ultimately able to test our initial assumption that two co-located VMs would have similar skew estimates.

### 3.3.1 Setup and Configuration

We set up our multi-server experimentation utilizing a VMWare ESXi server cluster at NPS. With the same Linux Ubuntu OS image as in the single-server experiments previously discussed, we built three Target VMs on two Dell PowerEdge R610 servers (8 x Intel(R)

Xeon(R) CPU E5620 @ 2.40 GHz, 100 GB RAM). This setup created one known pair of co-located VMs on one server and the third VM hosted by a separate server.

We configured the Dell laptop from the single-server experiments as our Data Collector node in order to provide a constant frame of reference in which to compare clock drift for each Target VM. As with the earlier experiments, the same Python scripts to generate TCP traffic were executed with the server script loaded on each of the three VMs and the client script run on the Data Collector. We executed the packet capture program `TCPDump` on the Data Collector during each test run in order to capture all of the TCP packets generated for parsing and skew analysis.

Our experiments were conducted through two separate tests, collecting timestamps from the two co-located VMs and then collecting timestamps from two VMs separately hosted. The reason for separate tests was to better isolate and study the results of the skew estimations for each test scenario. Each test consisted of ten trial runs, with 600 packets collected for each trial run. This sample size was chosen for the same reasons explained in the previous section. Each trial run collected the packets from each VM sequentially (i.e., collecting 600 packets from the one VM and then collecting 600 packets from the second). This is was done in order to minimize the influence of possible queuing and processing delays from the Data Collector conducting multiple TCP conversations and running parallel client programs.

### 3.3.2 Results

Our results for the two test sets were very promising. Ultimately, we were able to support our primary hypothesis that co-located VMs have similar skews. Figure 3.11 depicts the results from the first of ten trial runs for each of the two tests using the Wave Rider estimation model. The top two graphs in the figure depict the skew estimates from the two VMs co-located on the same physical server. While the skews are not an exact match, the relative error ($Error = \left|\frac{Skew_{VM_1} - Skew_{VM_2}}{Skew_{VM_1}}\right|$) between the two show the estimates are close in value (VM #2 has a relative error value of less than 5 percent with respect to VM #1). The difference between the two skew estimates can be explained through the linear regression model estimation and random network delays. The bottom two graphs depict the skew estimates for the second test, VMs that are not co-located. The skew estimates for the two

VMs are drastically different from each other (VM #2 has a relative error value of more than 280 percent with respect to VM #1). This result was replicated throughout every trial run in the two test scenarios. Since most data centers and cloud providers, to include AWS, deploy their VMs on Type I hypervisors, we can assume with some degree of confidence that our primary hypothesis holds validity in a real-world cloud architecture.

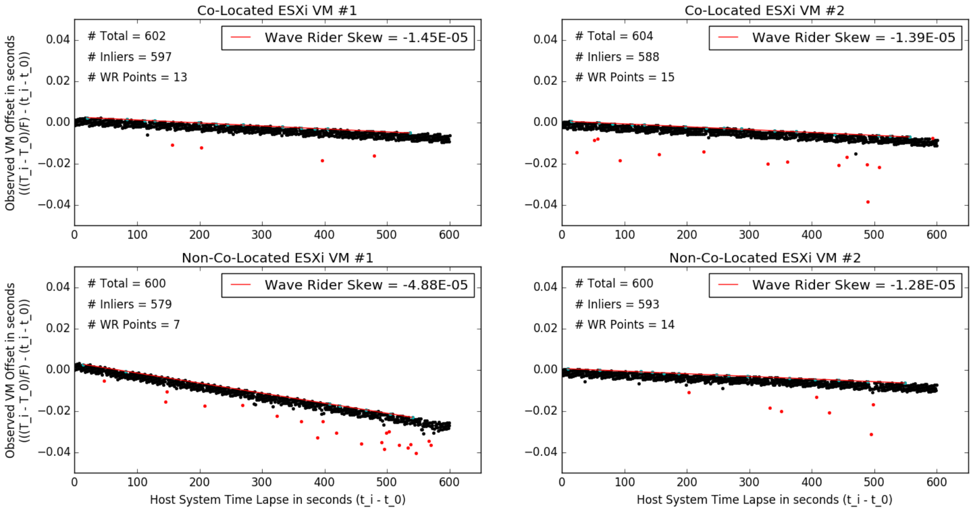


Figure 3.11: Results of the Time Skews Conducted on ESXi Type I Hypervisor for Both Co-Located (top row) and Non-Co-Located VM Trials(bottom row)

The results from each trial run for the two tests is listed in Table 3.6. Relative Error for each pair of VMs is calculated. The skew estimates for each VM individually over the ten trial runs identified an additional trend in the data. Specifically, the skew estimates for each VM were consistent throughout each test as each trial run estimate remained within 10 percent of the first trial run's value. The importance of this observation is the support it provides for one of our opening assumptions about using linear regression models to estimate clock skews: clock skew is constant over time. Extrapolating this trend, we argue that it does not matter when samples are collected from a VM. Any ten-minute window of packet capturing should return a skew estimate that is similar to any other ten-minute window. Ultimately, this insight will help determine the eventual TCP packet capture procedure in the AWS cloud.

Table 3.6: Multiple Server Skew Estimate Results

| | Test No. 1 (Co-located VMs) | | | Test No. 2 (VMs Not Co-located) | | |
|---|---|---|---|---|---|---|
| Trial No. | VM #1 | VM #2 | Relative Error | VM #1 | VM #2 | Relative Error |
| 1 | -1.45E-05 | -1.39E-05 | 0.043 | -4.88E-05 | -1.28E-05 | 281.0 |
| 2 | -1.45E-05 | -1.41E-05 | 0.028 | -4.80E-05 | -1.26E-05 | 281.0 |
| 3 | -1.38E-05 | -1.31E-05 | 0.053 | -4.87E-05 | -1.33E-05 | 266.0 |
| 4 | -1.28E-05 | -1.38E-05 | 0.073 | -4.88E-05 | -1.30E-05 | 275.0 |
| 5 | -1.43E-05 | -1.43E-05 | 0.000 | -4.88E-05 | -1.31E-05 | 273.0 |
| 6 | -1.35E-05 | -1.37E-05 | 0.015 | -4.85E-05 | -1.30E-05 | 273.0 |
| 7 | -1.37E-05 | -1.38E-05 | 0.007 | -4.85E-05 | -1.34E-05 | 262.0 |
| 8 | -1.41E-05 | -1.37E-05 | 0.029 | -4.86E-05 | -1.30E-05 | 274.0 |
| 9 | -1.44E-05 | -1.43E-05 | 0.007 | -4.90E-05 | -1.31E-05 | 274.0 |
| 10 | -1.46E-05 | -1.39E-05 | 0.014 | -4.89E-05 | -1.29E-05 | 279.0 |

## 3.4 Timestamp Simulation

In this section, we conduct a two-fold investigation. Shifting our collection method from a cooperative passive-collection process (having the Target periodically generate and send TCP packets to the Data Collector upon establishing a connection), we executed a Python script to simulate timestamp values using an uncooperative active-collection approach (requiring the Data Collector to periodically request a packet from the Target) that more closely reflects how a malicious cloud user would test for co-location in a cloud environment. We first analyzed the effect that various network traffic models had on the analysis of a known clock skew value, primarily determining the skew estimates degree of error from the true clock skew value. Secondly, we wanted to determine if our skew estimators behaved similarly in each traffic model or if some models outperformed the others in a given traffic scenario. We use four different delay models in our simulation scenarios. The first scenario generates packets with a constant delay value; however, since delays are never truly constant, this model serves strictly as a baseline to which the other models are compared to. The second scenario generates packets with normally-distributed delay values and the third scenario generates packets with exponentially-distributed delay values. Both of these models are used to establish performance envelopes for the skew estimators. However, as An *et al.* [30] described in his study of a self-similar network traffic model, real network traffic tends to follow heavy-tail patterns and act in a self-similar manner. In order to more accurately simulate this behavior, our fourth scenario utilizes delay values taken from live Internet traces into the AWS cloud.

### 3.4.1 Setup and Configuration

We executed our investigation within the Python 3.5 environment. Drawing from our understanding of TSval and typical packet routing behavior, we were able to generate 600 timestamps representing TSvals encoded in the TCP packet's TSopt header sent from the Target and another 600 timestamps reflecting the time the Data Collector received a packet from the Target. The simulated Target timestamps included an arbitrarily assigned constant drift value of 1.23E-05 seconds. We made a few assumptions regarding network characteristics in order to simplify our calculations and isolate the impact on the skew estimation by traffic delays shaped by our four models. Our assumptions were as follows:

- A uniform packet size of 10 Kb and a data transmission rate of 1 Mbps given for both Target and Data Collector, which provided a constant transmission delay of 0.01 seconds.
- OS processing delays for both Target and Data Collector are negligible and overtaken by larger network delays.
- Symmetrical travel times for ICMP packet round-trip delay values.
- No packets were dropped or re-sent.

With the queuing delay representing the delay variable in each model, we chose a value of 0.01 seconds for the constant delay scenario. To standardize the results, 0.01 seconds was also the mean delay value for both the randomly generated normal and exponential distributions. To act as a baseline comparison, we conducted one simulation trial with the constant delay scenario since results would not change with subsequent test runs. We conducted ten trial runs on both the normal and exponential models.

For the trace file scenario, we launched three instances into the AWS GovCloud (t2.micro: 1 virtual CPU burstable to 3.3 GHz, 1 GB Memory) and sent 600 ping requests to each VM's assigned Public IP address from our MacBook Pro laptop hosted on the NPS network. Each ping was sent after a one-second delay from the previous ping in order to simulate our packet collection method. This procedure was executed continuously for 10 hours to each VM instance in order to capture the fluctuating traffic volume levels experienced throughout a typical day, providing us with 30 different trace files of round trip times (RTTs). We derived queuing delay values by dividing each RTT value in half, then inserting them into our Python script as the delay component in the Target timestamp.

### 3.4.2 Results

Estimating the clock skew with all five estimation methods, the Constant Delay scenario generated skew estimates that were not as close to the true skew value as we anticipated. In order to better determine how close the skew estimates truly are, we calculated an error value for each estimator in relation to the true skew value of 1.23E-05 seconds ($Error = \left|\frac{Skew_{True} - Skew_{Estimate}}{Skew_{True}}\right|$). Of the five estimators, the Moving Average estimator outperformed all other models (Error = 0.12) while Wave Rider had an error almost four times greater than Moving Average (Error = 0.45). The results from the Normalized Delay scenario showed similar results to the Constant Delay scenario. The skew estimates remained lower than anticipated with the Moving Average estimator generating a mean error value of 0.12 over 10 trials while Wave Rider's mean error value increased to 0.84. The Exponential Delay scenario generated larger skew error values over all five estimators with Moving Average still outperforming the other models.

Our final scenario, inserting delay values from a trace file, generated the best overall performance from our estimators, with the exception of Wave Rider. The best performer in this scenario was RANSAC, as it had both the lowest mean error value and smallest standard deviation between trial run results. Both the Moving Average and OLS estimators performed well with mean error values below 0.10. Wave Rider was inconsistent as estimated skews varied between positive and negative values while having the largest standard deviation of skew estimates. Table 3.7 displays the results for all 30 trial runs from the cloud trace files.

Graphical results of the four scenarios showed a picture vastly different from results in previous sections. Illustrated in Figure 3.12, we found the data points plotted not as a band but more akin to a step function as clock offset values remain constant for a period of time before increasing in value. In addition, outlier data points were found above the inliers, as opposed to below them. This graphical behavior is most likely due to the change in packet collections. Previous tests used a cooperative passive-collection method between the Target and Data Collector such that once a TCP connection was established, the Target would send packets to the Data Collector in one-second intervals without any prompting. In this case, all delay values influenced the Data Collector timestamps, with large delays driving the clock offset value lower and minimum delays creating the upper bound limit. While this method was helpful in the establishment of our estimation methods, it is not a true representation of how packets would truly be collected. With the uncooperative active-collection method in these simulations, the Data Collector continuously requested a packet from the Target in one-second intervals, ultimately changing the location of the outliers from below the inliers to above them since network delay effects influenced both the Target timestamps and Data Collector timestamps. This behavior also explains the unreliability of Wave Rider, which is derived from the concept of riding the upper bound limit of data points. Selecting data points with clock offset values higher than then the points before and after, Wave Rider generated a regression line from selected outlier data points, which resulted in skew estimates that were generally not close to the true skew value.

Table 3.7: Skew Estimate Errors for Trace Delay Scenario

| | OLS | | Theil-Sen | | RANSAC | | Moving Average | | Wave Rider | |
|---|---|---|---|---|---|---|---|---|---|---|
| Trial No. | Skew | Error | Skew | Error | Skew | Error | Skew | Error | Skew | Error |
| 1 | 1.33E-05 | 0.08 | 1.50E-05 | 0.22 | 1.16E-05 | 0.06 | 1.28E-05 | 0.04 | 1.04E-03 | 83.55 |
| 2 | 1.30E-05 | 0.06 | 1.21E-05 | 0.02 | 1.06E-05 | 0.14 | 1.32E-05 | 0.07 | 3.19E-05 | 1.59 |
| 3 | 1.01E-05 | 0.18 | 1.03E-05 | 0.16 | 1.18E-05 | 0.04 | 1.00E-05 | 0.19 | 1.14E-04 | 8.27 |
| 4 | 1.32E-05 | 0.07 | 1.25E-05 | 0.02 | 1.20E-05 | 0.02 | 1.32E-05 | 0.07 | 1.11E-04 | 8.02 |
| 5 | 1.31E-05 | 0.07 | 8.39E-06 | 0.32 | 1.14E-05 | 0.07 | 1.29E-05 | 0.05 | 2.86E-05 | 1.33 |
| 6 | 1.18E-05 | 0.04 | 1.10E-05 | 0.11 | 1.35E-05 | 0.10 | 1.16E-05 | 0.06 | -1.67E-05 | 2.36 |
| 7 | 1.04E-05 | 0.15 | 9.09E-06 | 0.26 | 1.13E-05 | 0.08 | 1.10E-05 | 0.11 | -9.47E-05 | 8.70 |
| 8 | 1.22E-05 | 0.01 | 1.28E-05 | 0.04 | 1.21E-05 | 0.02 | 1.17E-05 | 0.05 | -6.32E-05 | 6.14 |
| 9 | 1.08E-05 | 0.12 | 7.14E-06 | 0.42 | 1.07E-05 | 0.13 | 1.07E-05 | 0.13 | -2.76E-05 | 3.24 |
| 10 | 1.17E-05 | 0.05 | 8.45E-06 | 0.31 | 1.09E-05 | 0.11 | 1.15E-05 | 0.07 | 1.15E-05 | 0.07 |
| 11 | 1.28E-05 | 0.04 | 7.38E-06 | 0.40 | 1.38E-05 | 0.12 | 1.16E-05 | 0.06 | -1.81E-04 | 15.72 |
| 12 | 1.25E-05 | 0.02 | 1.04E-05 | 0.15 | 1.29E-05 | 0.05 | 1.19E-05 | 0.03 | 1.66E-05 | 0.35 |
| 13 | 1.18E-05 | 0.04 | 5.67E-06 | 0.54 | 1.36E-05 | 0.11 | 1.17E-05 | 0.05 | -1.55E-04 | 13.60 |
| 14 | 1.16E-05 | 0.06 | 8.86E-06 | 0.28 | 1.17E-05 | 0.05 | 1.18E-05 | 0.04 | 7.37E-05 | 4.99 |
| 15 | 1.05E-05 | 0.15 | 2.53E-05 | 1.06 | 1.11E-05 | 0.10 | 1.05E-05 | 0.15 | 6.01E-05 | 3.89 |
| 16 | 1.30E-05 | 0.06 | 1.52E-05 | 0.24 | 1.32E-05 | 0.07 | 1.32E-05 | 0.07 | -4.56E-05 | 4.71 |
| 17 | 1.73E-05 | 0.41 | 1.98E-05 | 0.61 | 1.54E-05 | 0.25 | 1.76E-05 | 0.43 | -4.48E-06 | 1.36 |
| 18 | 8.20E-06 | 0.33 | 1.32E-05 | 0.07 | 1.26E-05 | 0.02 | 9.98E-06 | 0.19 | -3.13E-04 | 26.45 |
| 19 | 1.14E-05 | 0.07 | 9.45E-06 | 0.23 | 1.12E-05 | 0.09 | 1.10E-05 | 0.11 | 1.12E-05 | 0.09 |
| 20 | 9.66E-06 | 0.21 | 1.17E-05 | 0.05 | 1.07E-05 | 0.13 | 9.53E-06 | 0.23 | 7.20E-06 | 0.41 |
| 21 | 5.13E-06 | 0.58 | 9.60E-06 | 0.22 | 7.29E-06 | 0.41 | 3.89E-06 | 0.68 | -1.78E-05 | 2.45 |
| 22 | 1.28E-05 | 0.04 | 6.85E-06 | 0.44 | 1.49E-05 | 0.21 | 1.16E-06 | 0.91 | -4.04E-05 | 4.28 |
| 23 | 1.23E-05 | 0.00 | 6.17E-06 | 0.50 | 1.35E-05 | 0.10 | 1.24E-05 | 0.01 | -9.61E-05 | 8.81 |
| 24 | 1.40E-05 | 0.14 | 1.68E-05 | 0.37 | 1.42E-05 | 0.15 | 1.38E-05 | 0.12 | -1.78E-05 | 2.45 |
| 25 | 1.06E-05 | 0.14 | 1.36E-05 | 0.11 | 1.06E-05 | 0.14 | 8.90E-06 | 0.28 | -2.06E-05 | 2.67 |
| 26 | 1.48E-05 | 0.20 | 1.08E-05 | 0.12 | 1.38E-05 | 0.12 | 1.46E-05 | 0.19 | -3.62E-05 | 3.94 |
| 27 | 8.01E-06 | 0.35 | 1.15E-05 | 0.07 | 1.16E-05 | 0.06 | 1.07E-05 | 0.13 | 1.16E-16 | 1.00 |
| 28 | 1.36E-05 | 0.11 | 8.51E-06 | 0.31 | 1.36E-05 | 0.11 | 1.35E-05 | 0.10 | 5.79E-05 | 3.71 |
| 29 | 4.04E-06 | 0.67 | 9.29E-06 | 0.24 | 1.31E-05 | 0.07 | 8.91E-07 | 0.93 | -1.12E-03 | 92.06 |
| 30 | 1.36E-05 | 0.11 | 1.19E-05 | 0.03 | 1.39E-05 | 0.13 | 1.34E-05 | 0.09 | -1.63E-04 | 14.25 |
| Std. Dev. | 2.67E-06 | 0.16 | 4.15E-06 | 0.22 | 1.65E-06 | 0.08 | 3.53E-06 | 0.24 | 2.97E-04 | 21.68 |
| Mean | 1.16E-05 | 0.15 | 1.13E-05 | 0.26 | 1.23E-05 | 0.11 | 1.10E-05 | 0.19 | -2.83E-05 | 11.02 |

This table provides skew estimation error values for each estimator as compared to the known skew value of 1.23E-05 seconds for all 30 trial runs.

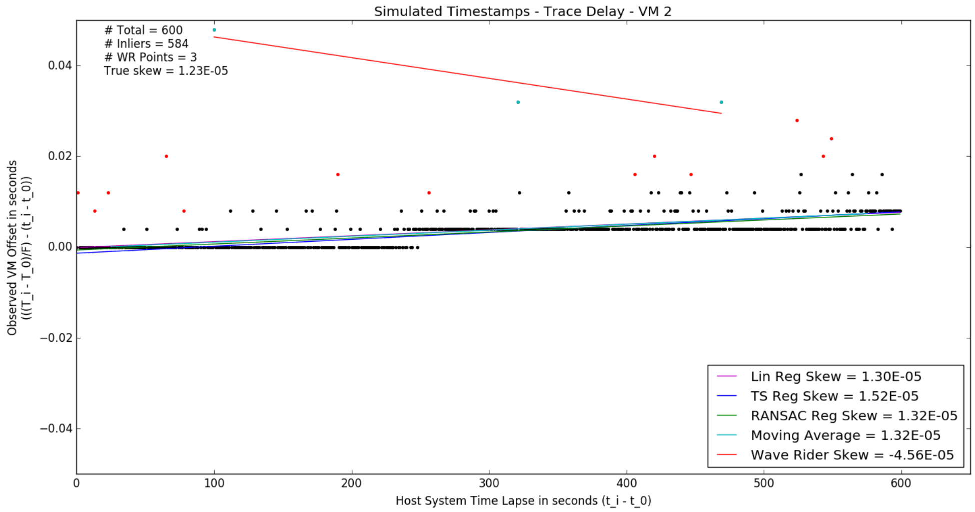


Figure 3.12: Simulated Timestamps–Trace Delay

### 3.4.3 Discussion

Overall, we discovered that the volume of traffic, in particular the network delay values, does have an impact on the accuracy of the clock skew estimation. The more congested the network, the higher the estimated skew variance and the larger the degree of error. However, since normal and exponential distributions are not strong representations of actual network traffic behavior, the trace scenario results carry much more weight. With the traces of real packet transmissions, the delay values are directly impacted by actual network traffic conditions. The constant delay scenario showed very little variation from skew estimates, with errors driven directly by the truncation of clock time units from microseconds to clock ticks. In contrast, the delay values inserted from the trace files affected some skew estimates to have much larger error values than our baselined measurements from the constant delay scenario. For example, the OLS and RANSAC estimators had a baseline error value of 0.19 but had error rates with the trace scenario as high as 0.67 and 0.41, respectively. It is important to note that real world routing decisions are generally impacted by load balancing. Most routers have multiple connections and will route traffic along the shortest path, altering a routing path if there is congestion or a disconnected link. While this is common practice within the publicly routed Internet, routing inside a non-public architecture may not have the same routing considerations. It is possible that routing redundancies are not available which could allow packet queues to build up, or queue release could be psuedo-randomized in an effort to prevent network mapping. Ultimately, while the real Internet routes packets to ensure that there is very little effect from traffic congestion, non-public networks could impact the effects of perceived network congestion.

Regarding the estimators, RANSAC and OLS were ultimately the top performers with Wave Rider as the worst. While RANSAC emerged as the top performer in our trace delay scenario, which best represents real traffic of our four models, OLS was not far behind. Out of 30 trials, RANSAC only had three error values that were over 0.15 (Trial Nos. 17, 21, and 22). In other words, RANSAC generated a reasonably accurate skew estimate 90% of the time. Of note, there was a general lack of consistency across all estimators for each trial run. For example, Trial No. 21 had error values of 0.58, 0.22, 0.41, 0.68, and 2.45 for the five estimators. Overall, RANSAC and OLS generally had similar error values throughout the 30 trials. Unfortunately, RANSAC still has the drawback of requiring fine tuning the Residual Threshold parameter, potentially causing the skew analysis process to be both long and tedious. In that regard, OLS becomes a strong estimator as it is both consistent and easy to calculate.

We argue though that Wave Rider should not be discarded as a valid estimator. While it underperformed, as compared to our previous testing, the concept behind Wave Rider is still correct. The simulated testing illuminated a flaw in the algorithm such that the selection of data points by clock offset value only works if they are truly inliers. With an active-collection approach developed, we believe Wave Rider can fit a linear model to a few points with minimum delay values and generate a skew estimation that more closely reflects the true skew of the sampled device while being completely immune to outlier data, outperforming the other estimators in cloud testing.

## 3.5 Optimizing Wave Rider

This section describes the process of adjusting Wave Rider to better account for the active-collection approach simulated in the previous section. We begin by discussing the changes to the estimator algorithm. We then detail the results of the simulation tests and discuss its benefits as a valid clock skew estimator.

### 3.5.1 Setup and Configuration

Adjusting for the more realistic active-collection approach, the Wave Rider Estimator was modified in two ways. The first modification has the algorithm account for minimum clock offset values instead of the previously used maximum values. Since the passive-collection approach could not determine packet delay values, Wave Rider estimated the clock skew based off data points with higher relative clock offset values than those before and after it with the belief that these data points reflected the packets with minimum network delay. With the higher clock offset values becoming outliers, the switch to minimum values should produce reliable estimates. The second modification accounts for the delays experienced by each packet and fits a regression line to a set number of data points with the smallest delay values. We conducted tests with the number of minimum-delay data points $n$ belonging to the set $n = \{2, 3, 5, 10, 15, 20, 25, 50\}$. We kept the numbers relatively small since the methodology behind Wave Rider is to only use a small subset of the collected data points that form a natural upper bound for the rest of the data points. We ran 30 tests with the same trace file delay scenario described in Section 3.4 for each Wave Rider configuration (a total of nine different configurations).

### 3.5.2 Results

The two modifications of the Wave Rider Estimator addressing the active-collection approach demonstrated overall improved performance. The resultant skew estimations and relative error (with respect to the known clock skew value of 1.23E-06 seconds) for each trial run are shown in Table 3.8 along with the calculated standard deviation of the skew estimates to determine consistency and the average skew estimation. The first modification, identifying minimal clock offset values as regression data points, illustrated the estimator's characteristic of ignoring the influence of outlier data. As illustrated in Figure 3.13, Wave Rider generated close skew estimates and outperformed or performed as well as the other estimators for the majority of the trials. However, the average skew estimate over 30 trials was still not as close to the true skew value than the other four models tested in the previous section.

Table 3.8: Skew Estimate Errors for Modified Wave Rider Estimator

| | Minimum Offsets | | Min Delay - 2 | | Min Delay - 3 | | Min Delay - 5 | | Min Delay - 10 | | Min Delay - 15 | | Min Delay - 20 | | Min Delay - 25 | | Min Delay - 50 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Trial No. | Skew | Error | Skew | Error | Skew | Error | Skew | Error | Skew | Error | Skew | Error | Skew | Error | Skew | Error | Skew | Error |
| 1 | 8.97E-06 | 0.27 | 1.18E-05 | 0.04 | 1.14E-05 | 0.07 | 8.75E-06 | 0.29 | 9.07E-06 | 0.26 | 9.19E-06 | 0.25 | 8.95E-06 | 0.27 | 1.07E-05 | 0.13 | 1.15E-05 | 0.07 |
| 2 | 1.32E-05 | 0.07 | 1.22E-05 | 0.01 | 1.24E-05 | 0.01 | 1.25E-05 | 0.02 | 5.47E-06 | 0.56 | 7.18E-06 | 0.42 | 7.85E-06 | 0.36 | 9.11E-06 | 0.26 | 1.11E-05 | 0.10 |
| 3 | 7.71E-06 | 0.37 | 9.46E-06 | 0.23 | 7.95E-06 | 0.35 | 9.83E-06 | 0.20 | 9.99E-06 | 0.19 | 1.07E-05 | 0.13 | 1.07E-05 | 0.13 | 1.10E-05 | 0.11 | 1.17E-05 | 0.05 |
| 4 | 2.29E-05 | 0.86 | 1.52E-05 | 0.24 | 1.59E-05 | 0.29 | 1.33E-05 | 0.08 | 1.03E-05 | 0.16 | 8.83E-06 | 0.28 | 1.09E-05 | 0.11 | 1.21E-05 | 0.02 | 1.25E-05 | 0.02 |
| 5 | 7.00E-06 | 0.43 | 4.23E-17 | 1.00 | 1.30E-05 | 0.06 | 9.47E-06 | 0.23 | 6.58E-06 | 0.47 | 9.28E-06 | 0.25 | 1.00E-05 | 0.19 | 9.63E-06 | 0.22 | 9.99E-06 | 0.19 |
| 6 | 8.32E-06 | 0.32 | 1.33E-05 | 0.08 | 1.84E-05 | 0.50 | 1.33E-05 | 0.08 | 1.27E-05 | 0.03 | 1.28E-05 | 0.04 | 1.25E-05 | 0.02 | 1.19E-05 | 0.03 | 1.23E-05 | 0.00 |
| 7 | 9.93E-06 | 0.19 | 0.00E+00 | 1.00 | 7.19E-05 | 4.85 | 1.23E-05 | 0.00 | 1.06E-05 | 0.14 | 1.02E-05 | 0.17 | 1.03E-05 | 0.16 | 1.00E-05 | 0.19 | 1.01E-05 | 0.18 |
| 8 | 1.44E-05 | 0.17 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 1.26E-05 | 0.02 | 1.16E-05 | 0.06 | 1.43E-05 | 0.16 | 1.28E-05 | 0.04 | 1.31E-05 | 0.07 | 1.31E-05 | 0.07 |
| 9 | 1.13E-05 | 0.08 | 8.64E-06 | 0.30 | 8.90E-06 | 0.28 | 8.20E-06 | 0.33 | 8.24E-06 | 0.33 | 8.44E-06 | 0.31 | 8.37E-06 | 0.32 | 8.33E-06 | 0.32 | 8.47E-06 | 0.31 |
| 10 | 1.47E-05 | 0.20 | 9.09E-06 | 0.26 | 8.05E-06 | 0.35 | 7.63E-06 | 0.38 | 6.70E-06 | 0.46 | 8.89E-06 | 0.28 | 9.07E-06 | 0.26 | 8.80E-06 | 0.28 | 9.52E-06 | 0.23 |
| 11 | 8.85E-06 | 0.28 | 0.00E+00 | 1.00 | 1.09E-05 | 0.11 | 1.27E-05 | 0.03 | 1.29E-05 | 0.05 | 1.20E-05 | 0.02 | 1.23E-05 | 0.00 | 1.20E-05 | 0.02 | 1.24E-05 | 0.01 |
| 12 | 1.70E-05 | 0.38 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 9.38E-06 | 0.24 | 1.22E-05 | 0.01 | 1.29E-05 | 0.05 | 1.21E-05 | 0.02 | 1.17E-05 | 0.05 | 1.10E-05 | 0.11 |
| 13 | 1.06E-05 | 0.14 | 9.96E-17 | 1.00 | 1.09E-05 | 0.11 | 1.24E-05 | 0.01 | 1.23E-05 | 0.00 | 1.11E-05 | 0.10 | 1.24E-05 | 0.01 | 1.15E-05 | 0.07 | 1.14E-05 | 0.07 |
| 14 | 7.52E-06 | 0.39 | 1.27E-05 | 0.03 | 9.45E-06 | 0.23 | 8.56E-06 | 0.30 | 8.98E-06 | 0.27 | 9.52E-06 | 0.23 | 9.53E-06 | 0.23 | 9.53E-06 | 0.23 | 1.03E-05 | 0.16 |
| 15 | 4.66E-17 | 1.00 | 1.56E-05 | 0.27 | 1.45E-05 | 0.18 | 1.71E-05 | 0.39 | 1.59E-05 | 0.29 | 1.39E-05 | 0.13 | 1.41E-05 | 0.15 | 1.27E-05 | 0.03 | 1.22E-05 | 0.01 |
| 16 | 0.00E+00 | 1.00 | 1.06E-05 | 0.14 | 1.01E-05 | 0.18 | 1.04E-05 | 0.15 | 9.54E-06 | 0.22 | 9.61E-06 | 0.22 | 1.14E-05 | 0.07 | 1.06E-05 | 0.14 | 9.89E-06 | 0.20 |
| 17 | 1.29E-05 | 0.05 | 9.11E-17 | 1.00 | 9.03E-17 | 1.00 | 7.09E-17 | 1.00 | 6.64E-17 | 1.00 | 7.14E-06 | 0.42 | 1.03E-05 | 0.16 | 1.00E-05 | 0.19 | 1.06E-05 | 0.14 |
| 18 | 2.13E-05 | 0.73 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 1.19E-16 | 1.00 | 1.08E-05 | 0.12 | 1.41E-05 | 0.15 | 1.25E-05 | 0.02 | 1.27E-05 | 0.03 | 1.14E-05 | 0.07 |
| 19 | 1.40E-05 | 0.14 | 2.67E-05 | 1.17 | 1.78E-05 | 0.45 | 1.89E-05 | 0.54 | 1.15E-05 | 0.07 | 9.90E-06 | 0.20 | 8.76E-06 | 0.29 | 9.25E-06 | 0.25 | 1.06E-05 | 0.14 |
| 20 | 1.20E-05 | 0.02 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 6.34E-06 | 0.48 | 8.09E-06 | 0.34 | 8.13E-06 | 0.34 | 8.26E-06 | 0.33 | 8.87E-06 | 0.28 | 9.24E-06 | 0.25 |
| 21 | 1.08E-05 | 0.12 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 3.35E-05 | 1.72 | 1.17E-05 | 0.05 | 9.18E-06 | 0.25 | 6.44E-06 | 0.48 | 5.59E-06 | 0.55 | 8.14E-06 | 0.34 |
| 22 | 1.19E-05 | 0.03 | 5.04E-17 | 1.00 | 1.17E-05 | 0.05 | 1.25E-05 | 0.02 | 1.38E-05 | 0.12 | 1.26E-05 | 0.02 | 1.33E-05 | 0.08 | 1.20E-05 | 0.02 | 1.27E-05 | 0.03 |
| 23 | 1.08E-05 | 0.12 | 0.00E+00 | 1.00 | 2.77E-05 | 1.25 | 1.66E-05 | 0.35 | 1.51E-05 | 0.23 | 1.47E-05 | 0.20 | 1.28E-05 | 0.04 | 1.35E-05 | 0.10 | 1.25E-05 | 0.02 |
| 24 | 0.00E+00 | 1.00 | 9.35E-06 | 0.24 | 9.95E-06 | 0.19 | 1.53E-05 | 0.24 | 1.55E-05 | 0.26 | 1.50E-05 | 0.22 | 1.49E-05 | 0.21 | 1.49E-05 | 0.21 | 1.49E-05 | 0.21 |
| 25 | 0.00E+00 | 1.00 | 1.82E-04 | 13.80 | 1.82E-04 | 13.80 | 8.36E-05 | 5.80 | 1.55E-05 | 0.26 | 1.23E-05 | 0.00 | 1.14E-05 | 0.07 | 8.91E-06 | 0.28 | 9.51E-06 | 0.23 |
| 26 | 1.08E-05 | 0.12 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 1.79E-05 | 0.46 | 1.89E-05 | 0.54 | 1.87E-05 | 0.52 | 1.62E-05 | 0.32 |
| 27 | 1.18E-05 | 0.04 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 1.47E-05 | 0.20 | 1.48E-05 | 0.20 | 1.61E-05 | 0.31 | 1.45E-05 | 0.18 | 1.43E-05 | 0.16 |
| 28 | 9.49E-06 | 0.23 | 0.00E+00 | 1.00 | 6.37E-17 | 1.00 | 3.77E-17 | 1.00 | 6.08E-17 | 1.00 | 6.47E-17 | 1.00 | 6.40E-17 | 1.00 | 6.50E-17 | 1.00 | 5.88E-17 | 1.00 |
| 29 | 1.37E-05 | 0.11 | 0.00E+00 | 1.00 | 0.00E+00 | 1.00 | 1.36E-05 | 0.11 | 1.21E-05 | 0.02 | 1.15E-05 | 0.07 | 1.24E-05 | 0.01 | 1.30E-05 | 0.06 | 1.29E-05 | 0.05 |
| 30 | 8.47E-06 | 0.31 | 1.49E-05 | 0.21 | 1.46E-05 | 0.19 | 1.49E-05 | 0.21 | 1.51E-05 | 0.23 | 1.57E-05 | 0.28 | 1.47E-05 | 0.20 | 1.41E-05 | 0.15 | 1.47E-05 | 0.20 |
| Std. Dev. | 5.48E-06 | 0.32 | 3.30E-05 | 2.43 | 3.41E-05 | 2.55 | 1.50E-05 | 1.07 | 4.45E-06 | 0.28 | 3.45E-06 | 0.19 | 3.39E-06 | 0.21 | 3.25E-06 | 0.20 | 2.84E-06 | 0.19 |
| Mean | 1.03E-05 | 0.34 | 1.17E-05 | 1.10 | 1.63E-05 | 1.12 | 1.31E-05 | 0.57 | 1.02E-05 | 0.28 | 1.11E-05 | 0.23 | 1.11E-05 | 0.20 | 1.10E-05 | 0.20 | 1.12E-05 | 0.16 |

This table provides skew estimation error values for each estimator as compared to the known skew value of 1.23E-05 seconds for all 30 trial runs. The numbers associated with the Min Delay columns represent the total data points selected to fit a regression line. Thus, “Min Delay - 5” used five data points correlating to the five smallest delay values.

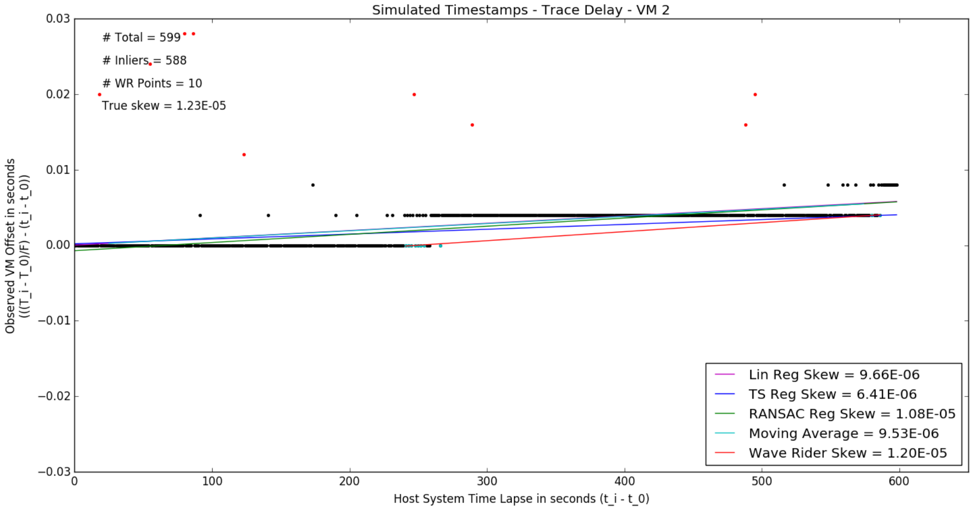


Figure 3.13: Simulated Timestamps–Trace Delay–Wave Rider with Miminum Clock Offset Values

The second modification made to the Wave Rider algorithm, selecting a number of $n$ data points whose network delay values are the lowest, showed greater improvement than the first modification. The average skew values over all 30 trials for each iteration of $n$ lowest-delay data points were consistently closer than the minimal clock offset adjustment. Figure 3.14 illustrates the estimation of the clock skew with $n = 5$ data points associated with the five lowest network delay values, once again ignoring the influence of outlier data points. A notable trend, as evidenced in Table 3.8, showed the average estimated clock skew initially got closer to the true value as the number of data points increased from 2 to 5 before decreasing in performance as the data points increased further. This was caused by a large concentration of data points in a specific area, driving the estimation to be biased. This performance is illustrated in Figure 3.15, where a concentration of data points in the first half of the graph influences the skew estimation further from the true value.

### 3.5.3 Discussion

Overall, the modifications to Wave Rider resulted in a solid performing estimator. Of the two adjustments, a skew estimate derived from the five data points associated with the five lowest network delay values performed the best. The model will never be influenced by outlier data points as data points with minimum delay values more closely reflect true

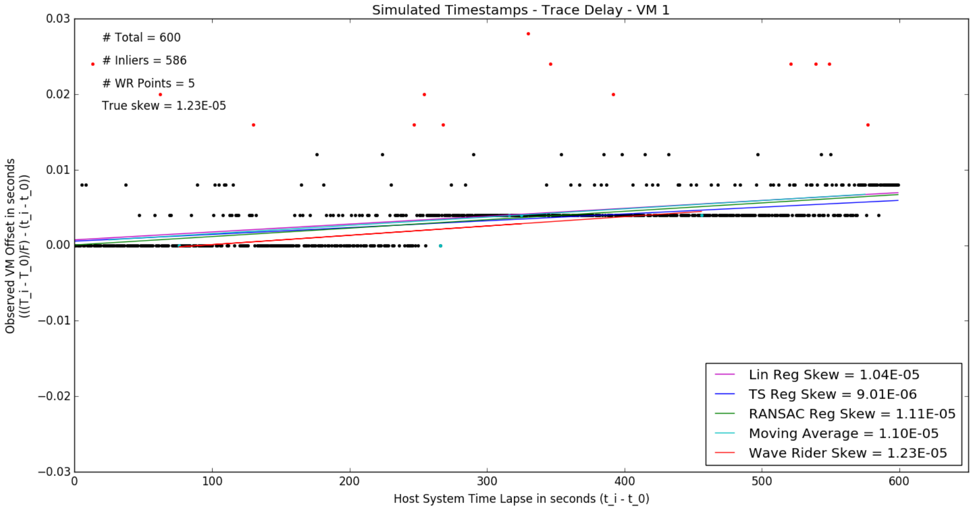


Figure 3.14: Simulated Timestamps–Trace Delay–Wave Rider with Minimum Delay Values

clock drift values and will always be found as an inlier. In addition, generating a linear regression from five data points helps to minimize the bias of concentrated minimal delay values. When past performance with live data collections is taken into consideration, we have shown that Wave Rider outperforms the other four estimators.

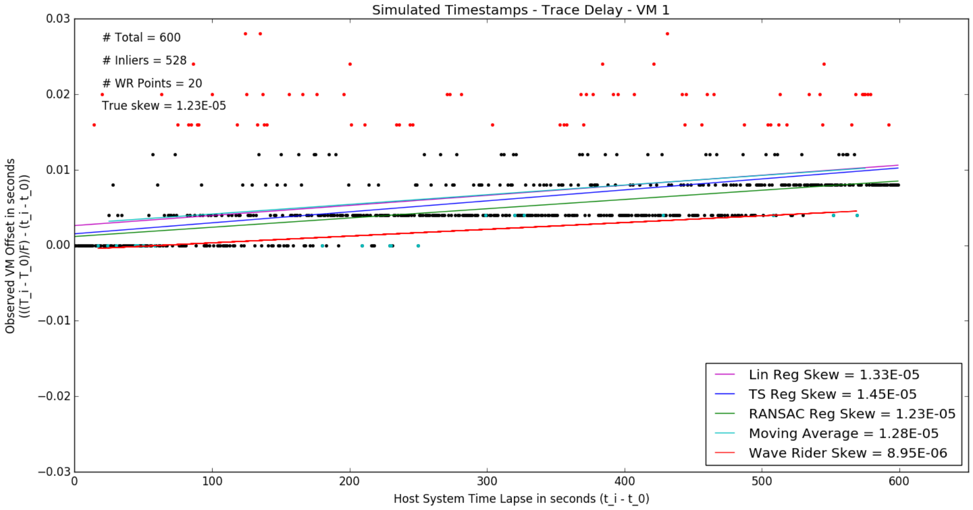


Figure 3.15: Simulated Timestamps–Trace Delay–Wave Rider with Minimum Delay Value Bias

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 4:
Public Cloud Experiments

In this chapter, we look to test our hypothesis in a public cloud environment. We begin by validating our testing methodology in the AWS GovCloud, ensuring that our determined minimum number of packets collected provides our estimators with enough data to return consistent results in a live setting and verifying the effectiveness of our estimators. We then deploy known co-located VMs and randomly located VMs to determine if the AWS cloud is susceptible to our co-location detection technique.

## 4.1 Validation of Cloud Testing Methods

Through our testing in Chapter 3, we developed a methodology with which to generate network traffic from a remote VM and estimate its clock skew. We discovered that collecting 600 TCP packets (approximately 10 minutes with one-second intervals between packet queries) was sufficient to properly estimate clock skew. In addition, we determined the skew estimator Wave Rider was the best performer of our five estimators after correcting for an active-collection approach to obtain TCP timestamps. In this section, we verify our testing parameters in a public cloud environment, validate the performance of Wave Rider, and investigate the behavior of clock skew estimates for a single public cloud VM over a period of time.

### 4.1.1 Setup and Configuration

After writing a Vagrant script to remotely launch VMs within the AWS cloud, we launched a single instance (m3.medium: 1 Virtual CPU, 3.75 GB Memory, 4 GB SSD storage, Amazon Linux OS) in the GovCloud Region. With the Data Collector from our controlled lab experiments, we ran Tshark (Wireshark's command line interface) and connected to the AWS instance with our traffic-generating script. We proceeded to collect 600 packets from the VM which were saved to a packet capture file, repeating this process until obtaining 30 packet capture files. Each capture file was then parsed, writing all Target ($T$) and Data Collector ($t$) timestamps as well as the RTTs for each of the 600 client-server interactions to a separate text file for skew estimation and analysis.

### 4.1.2 Results

Using our five estimation models on each trial run, we found results consistent with our previous experiments in Chapter 3. As shown in Table 4.1, Wave Rider generated the most consistent skew estimates of the five estimators. This observation was based off the relative error values for each trial run where $Error = \left|\frac{Skew_{Mean} - Skew_{Trial}}{Skew_{Mean}}\right|$ since the true clock skew value is unknown. The average error value and the standard deviation among the 30 samples for Wave Rider was 0.60 and 0.57 respectively, less then half the values for the OLS, RANSAC, and Moving Average estimators. The Theil-Sen estimator was the second lowest with an average error value and standard deviation of 0.76 and 0.75 respectively. Of note, while the other estimators generated both positive and negative skew estimates across the 30 samples, Wave Rider consistently generated negative skew estimates, with the exception of one trial run (Trial No. 19).

Visually, we observed that the behavior of Wave Rider had reverted back to our observations noted in Section 3.3, where the estimator “rode the waves” and delineated an upper bound limit while ignoring outlier data points instead of generating a lower bound limit noted in the simulated timestamp experiments within Section 3.5. An example of this is illustrated in Figure 4.1, where Wave Rider follows the general slope of the upper-boundary data points, correlated with minimal delay values. While the timestamp collection method is derived from the same active approach as in the timestamp simulations, the difference in the estimation behavior is attributed to the timestamp value $t$, which is referenced to the time of packet arrival at the Data Collector instead of packet departure from the Data Collector. These results were consistent through all 30 trials.

### 4.1.3 Discussion

Overall, the results of the Wave Rider Estimator through all 30 trials supports our finding that 600 packets provides the estimator with sufficient data to generate consistent results. While there was some variance from one trial run to another, Wave Rider’s estimates were usually between 5-10 $\mu s$ of the sample mean, a small margin compared to the other estimators. Wave Rider outperformed all other estimators because it was able to ignore the influence of data points driven by sporadic network delays. By generating a skew estimation from the five data points correlated to the five lowest delay values, Wave Rider strengthened both our argument that the upper-boundary data points most closely represent the true clock skew of

Table 4.1: Skew Estimate Errors for a Single Cloud VM

| | OLS | | Theil-Sen | | RANSAC | | Moving Average | | Wave Rider | |
|---|---|---|---|---|---|---|---|---|---|---|
| Trial No. | Skew | Error | Skew | Error | Skew | Error | Skew | Error | Skew | Error |
| 1 | -2.01E-05 | 0.02 | -7.91E-07 | 0.97 | -1.02E-05 | 0.17 | -2.32E-05 | 0.05 | -6.18E-06 | 0.61 |
| 2 | -4.92E-05 | 1.50 | -2.52E-05 | 0.02 | -3.95E-05 | 2.22 | -6.02E-05 | 1.47 | -3.77E-05 | 1.37 |
| 3 | -2.13E-05 | 0.08 | -4.02E-05 | 0.57 | -2.86E-05 | 1.33 | -2.17E-05 | 0.11 | -2.68E-05 | 0.69 |
| 4 | -9.17E-06 | 0.53 | -1.24E-05 | 0.52 | -4.85E-06 | 0.60 | -1.89E-05 | 0.22 | -3.57E-05 | 1.25 |
| 5 | 7.43E-06 | 1.38 | -1.37E-05 | 0.47 | -5.98E-06 | 0.51 | 1.06E-05 | 1.43 | -2.51E-05 | 0.58 |
| 6 | -1.54E-05 | 0.22 | -8.29E-06 | 0.68 | -1.43E-05 | 0.17 | -7.20E-06 | 0.70 | -1.74E-05 | 0.10 |
| 7 | -2.52E-05 | 0.28 | -1.16E-05 | 0.55 | -2.01E-05 | 0.64 | -2.73E-05 | 0.12 | -1.79E-05 | 0.13 |
| 8 | -9.48E-05 | 3.83 | -4.45E-05 | 0.73 | -3.51E-05 | 1.86 | -1.02E-04 | 3.18 | -3.14E-05 | 0.98 |
| 9 | -1.84E-04 | 8.37 | -1.08E-04 | 3.21 | -7.53E-05 | 5.14 | -2.15E-04 | 7.82 | -1.45E-05 | 0.09 |
| 10 | 1.64E-05 | 1.83 | -1.16E-05 | 0.55 | -6.61E-06 | 0.46 | 1.50E-05 | 1.62 | -9.83E-06 | 0.38 |
| 11 | 6.37E-05 | 4.24 | -8.10E-06 | 0.68 | 4.74E-07 | 1.04 | 5.81E-05 | 3.38 | -1.52E-05 | 0.04 |
| 12 | -6.45E-05 | 2.28 | -2.68E-05 | 0.04 | -3.81E-05 | 2.10 | -8.63E-05 | 2.54 | -2.59E-05 | 0.63 |
| 13 | 1.18E-05 | 1.60 | -7.77E-06 | 0.70 | -9.85E-07 | 0.92 | 2.97E-05 | 2.22 | -4.01E-06 | 0.75 |
| 14 | 5.97E-06 | 1.30 | -2.54E-05 | 0.01 | -2.22E-05 | 0.81 | 1.63E-05 | 1.67 | -1.40E-05 | 0.12 |
| 15 | 1.25E-05 | 1.64 | -3.15E-05 | 0.23 | -1.62E-05 | 0.32 | 1.35E-05 | 1.55 | -1.59E-05 | 0.00 |
| 16 | -7.25E-05 | 2.69 | -4.55E-05 | 0.77 | -3.33E-05 | 1.71 | -8.17E-05 | 2.35 | -2.04E-05 | 0.28 |
| 17 | -4.60E-05 | 1.34 | -6.69E-05 | 1.61 | -2.18E-05 | 0.78 | -6.36E-05 | 1.61 | -1.18E-05 | 0.26 |
| 18 | -3.02E-05 | 0.54 | -4.20E-06 | 0.84 | 8.07E-06 | 1.66 | -4.16E-05 | 0.71 | -1.37E-05 | 0.14 |
| 19 | -3.65E-05 | 0.86 | -5.44E-05 | 1.12 | -1.27E-05 | 0.03 | -1.91E-05 | 0.22 | 2.43E-05 | 2.53 |
| 20 | -3.40E-05 | 0.73 | -4.11E-05 | 0.60 | 1.74E-06 | 1.14 | -3.80E-05 | 0.56 | -1.00E-05 | 0.37 |
| 21 | -6.16E-06 | 0.69 | -2.76E-05 | 0.08 | 2.60E-05 | 3.12 | -1.46E-05 | 0.40 | -3.40E-05 | 1.14 |
| 22 | 1.19E-06 | 1.06 | -1.85E-05 | 0.28 | -2.14E-05 | 0.74 | -3.11E-07 | 0.99 | -7.47E-06 | 0.53 |
| 23 | -3.61E-06 | 0.82 | -5.14E-06 | 0.80 | -8.67E-06 | 0.29 | -7.83E-06 | 0.68 | -1.20E-05 | 0.24 |
| 24 | 7.95E-06 | 1.40 | -1.15E-05 | 0.55 | -2.82E-06 | 0.77 | 9.68E-06 | 1.40 | -8.31E-08 | 0.99 |
| 25 | -1.37E-05 | 0.30 | -1.16E-05 | 0.55 | -1.42E-05 | 0.16 | -2.30E-05 | 0.06 | -1.10E-06 | 0.93 |
| 26 | -5.70E-05 | 1.90 | -3.05E-05 | 0.19 | -2.69E-05 | 1.19 | -5.51E-05 | 1.26 | -1.19E-05 | 0.25 |
| 27 | -4.14E-05 | 1.11 | -4.52E-05 | 0.76 | -3.46E-05 | 1.82 | -3.99E-05 | 0.64 | -4.16E-05 | 1.62 |
| 28 | -5.83E-07 | 0.97 | -2.09E-05 | 0.19 | -2.79E-06 | 0.77 | -1.07E-05 | 0.56 | -2.07E-05 | 0.30 |
| 29 | 1.32E-04 | 7.72 | 5.31E-05 | 3.07 | 1.15E-04 | 10.37 | 1.03E-04 | 5.22 | -1.58E-05 | 0.01 |
| 30 | -2.29E-05 | 0.17 | -6.44E-05 | 1.51 | -2.22E-05 | 0.81 | -3.02E-05 | 0.24 | -2.75E-06 | 0.83 |
| Std. Dev. | 5.20E-05 | 1.99 | 2.77E-05 | 0.75 | 3.03E-05 | 1.98 | 5.52E-05 | 1.68 | 1.33E-05 | 0.57 |
| Mean | -1.96E-05 | 1.71 | -2.57E-05 | 0.76 | -1.23E-05 | 1.46 | -2.44E-05 | 1.50 | -1.59E-05 | 0.60 |

This table provides skew estimation error values for each estimator as compared to that estimator's mean skew value for all 30 trial runs in order to evaluate a measure of consistency for each estimator.

the VM and our argument that it is the estimator best suited to compare the skews of two VMs.

## 4.2 Analysis of Known Co-Located Instances

With a detailed testing methodology derived from previous experiments and a reliable skew estimation model in Wave Rider, we begin our final tests in a live cloud environment. In

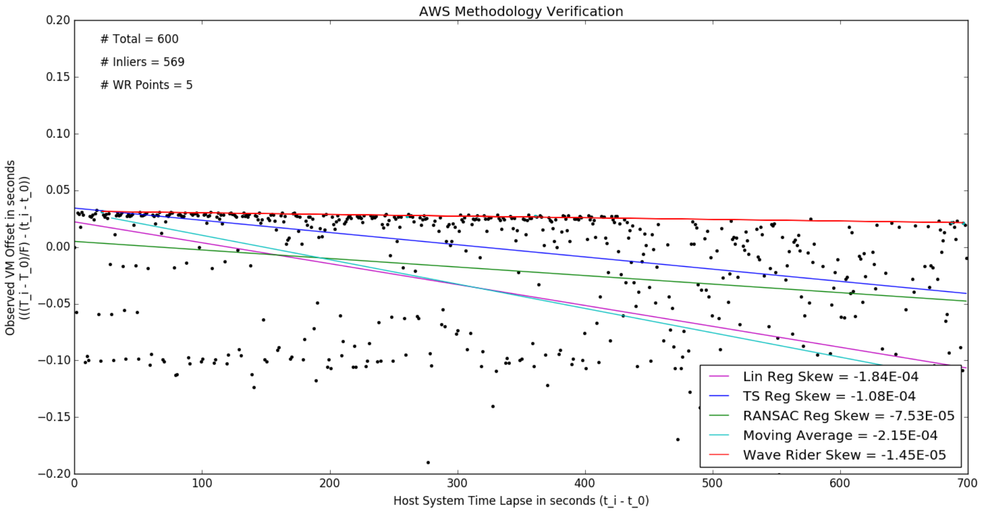


Figure 4.1: AWS Methodology Verification

this section, we first describe our environment configuration in the AWS cloud. We then discuss our results and determine if our VM co-location technique is applicable within the AWS cloud.

### 4.2.1 Cloud Configuration

Before we could begin launching EC2 instances into the cloud, we had to determine a way in which we could guarantee the co-location of two VMs with all other instances randomly placed on separate servers. Amazon's Dedicated Host service allowed us to leverage this capability for our test. A Dedicated Host is a physical server in an AWS Region that is reserved for the use of a single user, no other cloud user can launch an instance on this device [16]. Once allocated, the Dedicated Host is assigned a unique Host ID that is used to launch an instance to that specific server. Our test was conducted in the Northern California (US-West-1) Region instead of the GovCloud Region since Dedicated Hosts are not yet available there.

We began our test by launching two m3.medium instances onto shared servers, referred to as Hunter 2 and Hunter 3, prior to allocating a Dedicated Host. This ensured that neither VM would be co-located with the Target. We use the term *Hunter* to identify all adversary VMs that attempt to co-locate with the Target. Once the two Hunters and the Dedicated

Host were initialized, we launched two m3.medium instances (the Target and Hunter 1) onto the Dedicated Host. With the Apple MacBook Pro laptop as the Data Collector, we ran Tshark on the Data Collector and collected 600 packets from each instance in series with all packets saved to a packet capture file. This test sequence was continuously repeated until 10 trial runs were successfully completed.

### 4.2.2 Results

The skew estimation results for each trial is summarized in Table 4.2. A relative error value ($Error = \left|\frac{Skew_{Target} - Skew_{Hunter}}{Skew_{Target}}\right|$) was also calculated for each Hunter over all 10 trial runs. The mean and standard deviation for skew and relative error over all trial runs were then calculated for each VM. Trial Nos. 7-9 showed strong support for the co-location of Target and Hunter 1 while Hunter 2 and Hunter 3 were both suggested to be on separate servers from the Target based solely on the relative error values. The results from Trial No. 7 are illustrated in Figure 4.2. It should be noted that not all trial runs analyzed a full 600 collection packets for each VM. This was due to some packets either getting lost, sporadicly retransmitted packets, and duplicated acknowledgments. To correctly analyze the packet capture files, we filtered out only known "good" packets, discarding five packets per VM on average. Since the number of packets remained well above 500, we determined the adjusted sample size was still sufficient to generate reliable skew estimates, as supported from our findings in Section 3.1.

In contrast, generated results from Trial Nos. 2 and 10 suggested that Hunter 2 and Hunter 3 were both co-located with the Target based on the relative errors, as depicted in Figure 4.3. In both cases, the relative error value for Hunter 1 was more than twice the value of Hunter 2 or Hunter 3. In the other trials, the relative error value from Hunter 1 did not strongly support co-location with the Target as the value was either close or above the error values for Hunter 2 and Hunter 3.

When the mean skew estimates and error values for all three Hunter VMs are compared to the Target, Hunter 1 lacked support to claim co-location. This was mostly due to the effect of the skew estimate of Trial No. 6 which generated an error value of 45.58. When that skew estimate is removed and the nine remaining estimates and errors are averaged, the resultant values dropped down -6.80E-06 and 0.44 respectively. While closer to our expectations,

Table 4.2: Skew Estimate Errors Determining Co-Location in a Public Cloud

| | Target | Hunter 1 | | Hunter 2 | | Hunter 3 | |
|---|---|---|---|---|---|---|---|
| Trial No. | Skew | Skew | Error | Skew | Error | Skew | Error |
| 1 | 2.08E-05 | -1.00E-06 | 1.05 | -8.89E-06 | 1.43 | -7.13E-06 | 1.34 |
| 2 | -1.40E-05 | -8.33E-06 | 0.41 | -1.16E-05 | 0.17 | -1.38E-05 | 0.01 |
| 3 | -1.32E-05 | -9.75E-06 | 0.26 | -9.39E-06 | 0.29 | -7.33E-06 | 0.44 |
| 4 | -1.39E-05 | -4.70E-06 | 0.66 | -8.40E-06 | 0.40 | -7.74E-06 | 0.44 |
| 5 | -5.40E-06 | -2.33E-06 | 0.57 | -7.58E-06 | 0.40 | -9.21E-06 | 0.71 |
| 6 | -4.15E-06 | 1.85E-04 | 45.58 | -3.10E-06 | 0.25 | -1.14E-05 | 1.75 |
| 7 | -7.73E-06 | -7.33E-06 | 0.05 | 2.36E-07 | 1.03 | -1.38E-05 | 0.79 |
| 8 | -1.39E-05 | -1.65E-05 | 0.19 | -6.12E-06 | 0.56 | -5.61E-06 | 0.60 |
| 9 | -5.07E-06 | -5.24E-06 | 0.03 | -9.33E-06 | 0.84 | 6.59E-07 | 1.13 |
| 10 | -3.37E-06 | -6.01E-06 | 0.78 | -4.56E-06 | 0.35 | -2.23E-06 | 0.34 |
| Std. Dev. | 1.04E-05 | 6.08E-05 | 14.28 | 3.53E-06 | 0.40 | 4.65E-06 | 0.52 |
| Mean | -5.99E-06 | 1.24E-05 | 4.96 | -6.87E-06 | 0.57 | -7.76E-06 | 0.75 |

This table provides skew estimation error values for each Hunter as compared to the Target's skew value for all 10 trial runs.

the values were very close to Hunter 2.

### 4.2.3 Discussion

Overall, this test failed to support our hypothesis that clock skew estimation was a reliable method to detect co-location of VMs in the AWS environment. Hunter 1 failed to generate consistent skew estimates that were similar to the Target but dissimilar from both Hunter 2 and Hunter 3. Our earlier testing in Section 3.3 clearly demonstrated the difference in estimate values between co-located and separated VMs. Therefore, we believe the result of both Hunter 2 and Hunter 3 generating similar skew values to both the Target and Hunter 1, despite residing on separate servers, can be explained in one of two ways: First, Wave Rider was not sufficiently able to filter out all delay bias in the timestamp transformation. The scale of network delay was three orders of $\mu s$ greater than the clock skew. This caused an overshadow affect when parsing and transforming the timestamps for analysis. This effect was not seen with the NPS lab testing since routing between the VMs and the Data Collector was through on-premises switches, which resulted with vastly smaller delay times (< 1 ms vice 25 ms) and provided clear and consistent results. The testing in the AWS

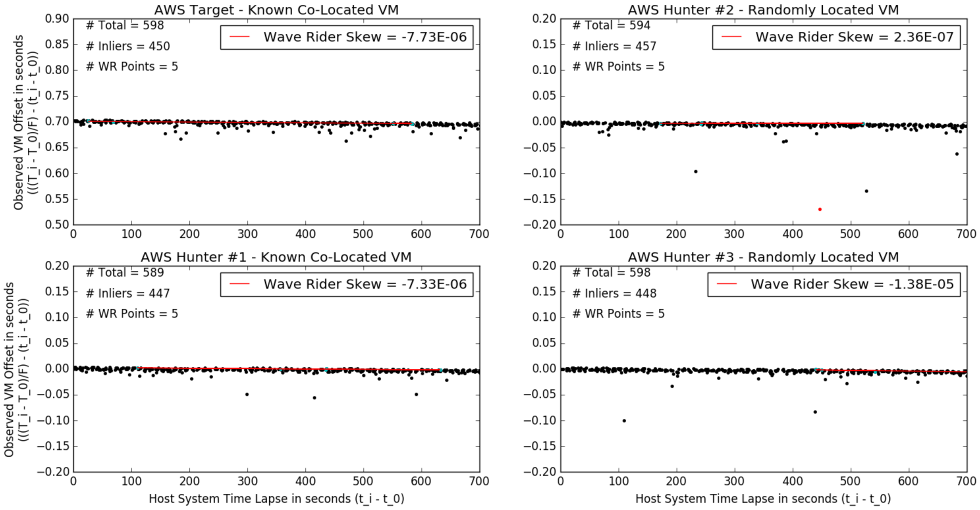


Figure 4.2: Public Cloud Test–Positive Result

cloud occurred over far greater distances, which resulted in much higher delay times and far less consistent results. Also, much of the previous work associated with cloud-mapping and VM co-location in Section 2.3.3 occurred within the AWS cloud, exploiting tools that relied on delay times and routing paths. In an effort to combat these techniques, it is likely that AWS purposely adjusted the release of packets to randomize delay values or even exploited their own architecture redundancies by randomizing the route path from the ToR/EoR switch to the gateway router, which resulted in higher delay values and inconsistent test results. Secondly, there might be some form of adjustment associated with AWS VMs timekeeping. This would obfuscate the clock skew estimation enough that there is no reliable way to determine whether two VMs are actually co-located, ultimately preventing successful application of this detection technique from the onset.

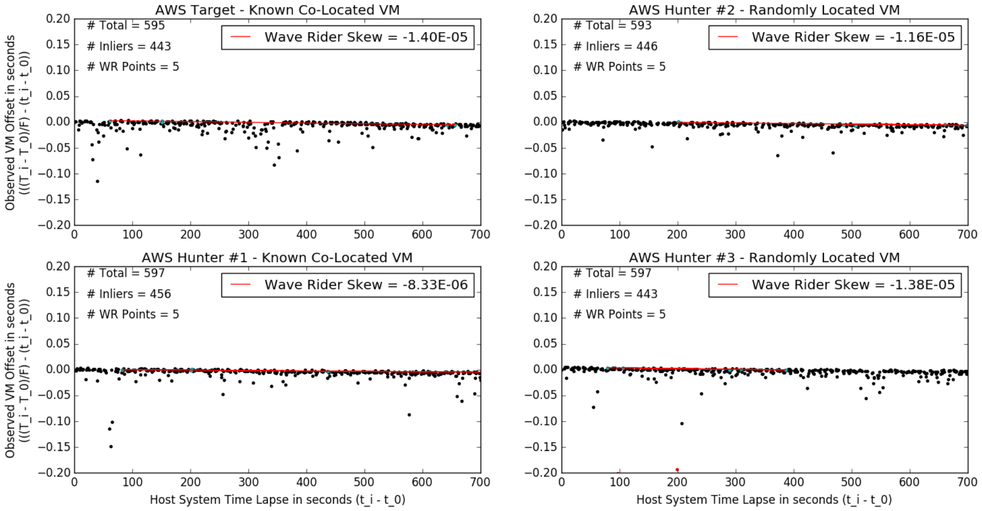


Figure 4.3: Public Cloud Test–Negative Result

# CHAPTER 5: Conclusion

In summary, we were able to learn a great deal about TCP timestamps, clock skews, VMs, and cloud security. Regarding the detection of VM co-location specifically, we were able to determine that a collection of at least 500 TCP timestamps will generate a reliable skew estimate (Section 3.1) from nearby Data Collectors (but not distant ones). We learned that while unmitigated network traffic congestion does impact the estimation of a VM's clock skew, in reality this should not largely influence skew estimations as mechanisms within TCP and redundancies in routing paths try to minimize delay effects (Section 3.4). Our largest contribution to the research of co-location detection, we created the Wave Rider Estimator, a new method for estimating clock skews. It fit a linear regression to the five points that correspond to the five smallest one-way latency values between the VM and the Data Collector to combat influence from sporadic network delays (Section 3.5). Lastly, we determined that the AWS public cloud is not susceptible to VM co-location detection via clock skew comparison with our tested methods when probing from outside the AWS cloud (Section 4.2). However, we believe that the negative results of this testing was most likely due to large, inconsistent network delays. We recommend any future work on this topic should look at configuring the Data Collector as a VM within the cloud, where routing delays should be greatly minimized which should result in more consistent and conclusive results.

In addition, due to time, personnel, and contractual constraints, we highly encourage others to extend our research to answer questions we could not. A short list includes:

- Do different hypervisors influence a VM's clock skew?
- What cloud providers are susceptible to VM co-location determination via clock skew estimation?
- What is the optimum number of data points Wave Rider needs?
- Can one-way packet delay times be accurately collected in order to improve the reliability of Wave Rider?
- Does the location of VM probing affect clock skew estimation results?

Our primary hypothesis was validated on the performance of a single Type I hypervisor (VMWare ESXi) as we wanted to test a product similar to Amazon's Xen-based hypervisor and we had no other hypervisor readily available. There are many other Type I hypervisors available, both open and closed source, that may be used by other cloud providers or business data centers which may or may not be susceptible to co-location detection via clock skew estimation. Additionally, our live cloud experiments focused solely on AWS due to contractual limitations. However, Google and Microsoft are also popular public cloud providers with configurations and architectures that may behave differently than AWS. While our primary research question was answered in the negative, it would be beneficial to learn if other public cloud providers are as resistant to this co-location technique as AWS is since the concept was proven to work in the NPS data center. In addition, launching a Data Collector into the cloud would reduce the network delays of collected packets and might provide more consistent results. Finally, we leave the fine tuning of Wave Rider, our biggest research contribution, to further work. Our final iteration of the estimator derived the "optimal" number of points from a single test scenario. While five data points appeared to work, both statistically and visually, we believe that more rigorous testing is required to determine a parameter value that optimizes the performance of Wave Rider. Also, the RTT values referenced by Wave Rider to determine the data points by which to generate a skew estimate from assumes symmetric delay times in each direction of travel. In the real world, however, a round trip is rarely symmetric and it is not uncommon to have a short delay in one direction and a long delay in the other. This behavior can very easily produce incorrect data points for skew estimation which could negatively influence the performance of Wave Rider. Determining how to identify and ignore the affects of delay values would ultimately strengthen Wave Rider.

The estimation of clock skews from TCP timestamps is a simple process; however, this technique can be countered by several methods. First, there are plenty of researched methods [8] to actively suppress, or minimize, a device's clock skew, such as routinely synchronizing with a Network Time Protocol (NTP) server. Secondly, users can disable the TCP TSopt on installed web browsers to prevent timestamps from getting encoded in each packet header. In addition, users can pay for Dedicated Hosts/servers in a cloud environment, to prevent any non-organizational user from launching VMs alongside theirs. Lastly, time could be adjusted at hypervisors, which could be encoded to purposely alter the clock skew of each hosted VM in order to obfuscate estimation results and generate false positives, or with routing decisions by randomizing the release of packets from switches and routers or the routing path itself.

In conclusion, while our testing within the AWS cloud environment rejected our hypothesis that clock skew comparison could determine VM co-location, we strongly argue that this result is due solely to the effects of large, inconsistent network delay values. Our testing on an NPS data center with consistent delay values supports the effectiveness of this technique and that it needs to be defended against. Arguably, the largest threat to the DOD is the insider threat. By having direct information to a specific VM, the insider can act directly, or indirectly by providing information to a third party, and begin deploying Hunter VMs in order conduct a co-location attack. Understanding whether the cloud provider and hypervisor can prevent co-location determination through clock skew comparison will ultimately lead to a better cyber-security strategy.

THIS PAGE INTENTIONALLY LEFT BLANK

# List of References

[1] A. L. Timmons, P. Fomin, and J. S. Wasek, “Modeling cloud storage: A proposed solution to optimize planning for and managing storage as a service,” *Journal of Information and Computing Science*, vol. 11, no. 1, pp. 70–80, 2016.

[2] L. Columbus. (2016). Roundup of cloud computing forecasts and market estimates, 2016. [Online]. Available: http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016/#1a32d5c474b0

[3] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 199–212.

[4] Amazon Virtual Private Cloud: User guide. (2016). Amazon Web Services. [Online]. Available: http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-ug.pdf

[5] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler, “Detecting co-residency with active traffic analysis techniques,” in *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*. ACM, 2012, pp. 1–12.

[6] A. Herzberg, H. Shulman, J. Ullrich, and E. Weippl, “Cloudoscopy: Services discovery and topology mapping,” in *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*. ACM, 2013, pp. 113–122.

[7] Z. Xu, H. Wang, and Z. Wu, “A measurement study on co-residence threat inside the cloud,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 929–944.

[8] T. Kohno, A. Broido, and K. C. Claffy, “Remote physical device fingerprinting,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 93–108, 2005.

[9] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. ACM, 2010, pp. 267–280.

[10] L. Wang, A. Nappa, J. Caballero, T. Ristenpart, and A. Akella, “WhoWas: A platform for measuring web deployments on IaaS clouds,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 101–114.

[11] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-tenant side-channel attacks in PaaS clouds,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 990–1003.

[12] M. J. Kavis, *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*, 1st ed. Hoboken, NJ: John Wiley and Sons, Inc., 2014.

[13] D. Perez-Botero, J. Szefer, and R. B. Lee, “Characterizing hypervisor vulnerabilities in cloud computing servers,” in *Proceedings of the 2013 International Workshop on Security in Cloud Computing*. ACM, 2013, pp. 3–10.

[14] K. K. Sheridan-Barbian, “A survey of real-time operating systems and virtualization solutions for space systems,” DTIC Document, Tech. Rep., 2015.

[15] Free virtualization software & hypervisors. (2015, Dec. 13). [Online]. Available: http://webtechmag.com/free-virtualization-software-hypervisors/

[16] Amazon Elastic Compute Cloud: User guide for linux instances. (2016). Amazon Web Services. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-ug.pdf

[17] C. Onwubiko, “Security issues to cloud computing,” in *Cloud Computing*. Springer, 2010, pp. 271–288.

[18] J. P. Durbano, D. Rustvold, G. Saylor, and J. Studarus, “Securing the cloud,” in *Cloud Computing*. Springer, 2010, pp. 289–302.

[19] C. Pfleeger and S. Pfleeger, *Analyzing Computer Security: A Threat/Vulnerability/Countermeasure Approach*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2012.

[20] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, “Thermal covert channels on multi-core platforms,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 865–880.

[21] S.-J. Moon, V. Sekar, and M. K. Reiter, “Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1595–1606.

[22] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, “A placement vulnerability study in multi-tenant public clouds,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 913–928.

[23] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 177–186.

[24] D. Borman, R. Scheffenegger, and V. Jacobson, “TCP extensions for high performance,” 2014.

[25] Enabling tcp timestamp linux and windows. (2013, Feb. 27). [Online]. Available: http://ithitman.blogspot.com/2013/02/enabling-tcp-timestamp-linux-and-windows.html

[26] G. Keller, *Statistics For Management and Economics*, 9th ed. Mason, OH: South-Western Cengage Learning, 2012.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[28] J. W. Osborne, *Best Practices in Quantitative Methods*. Sage, 2008.

[29] A. Hast, J. Nysjö, and A. Marchetti, “Optimal RANSAC-Towards a repeatable algorithm for finding the optimal set,” 2013.

[30] X. An, L. Qu *et al.*, “A study based on self-similar network traffic model,” in *2015 Sixth International Conference on Intelligent Systems Design and Engineering Applications (ISDEA)*. IEEE, 2015, pp. 73–76.

THIS PAGE INTENTIONALLY LEFT BLANK

# Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California